RasPi

DESIGN
BUILD
CODE

30

Get hands-on with your Raspberry Pi

+ BUILD A **PI CLUSTER**

PI ZERO

THE COMPLETE START-UP GUIDE

Plus Make a Zero-powered wearable

Welcome

The Raspberry Pi was a revolutionary piece of kit – clever, hackable, great value – but not content with that, Eben Upton and the rest of the crew decided they could do even better. Released in November 2015, the even-smaller-than-pocketsized Pi Zero packs in 512MB RAM, a 32-bit ARMv6 chip, and a low price of just $5. Amazing! But what's even more amazing is what you can do with it.

This issue, we'll show you how to set up the Pi Zero to get the most from it, and then dive into a project that lets you use the diminutive SBC as part of your very own wearable! Plus, learn how to combine multiple Pis into a cluster (who said supercomputing was just for supercomputers?) and much more. Enjoy!

April

Editor

Get inspired

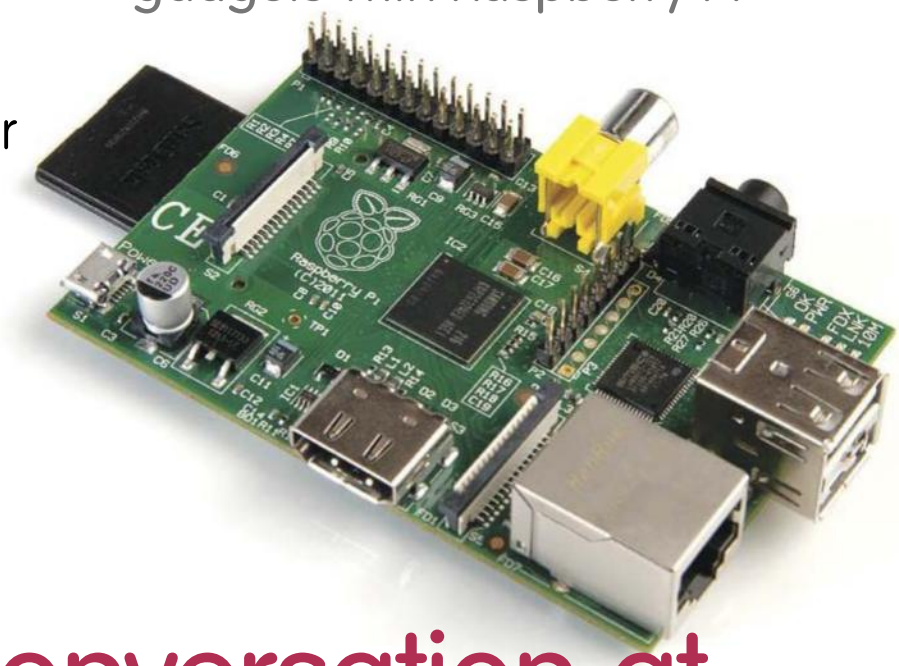
Discover the RasPi community's best projects

Expert advice

Got a question? Get in touch and we'll give you a hand

Easy-to-follow guides

Learn to make and code gadgets with Raspberry Pi

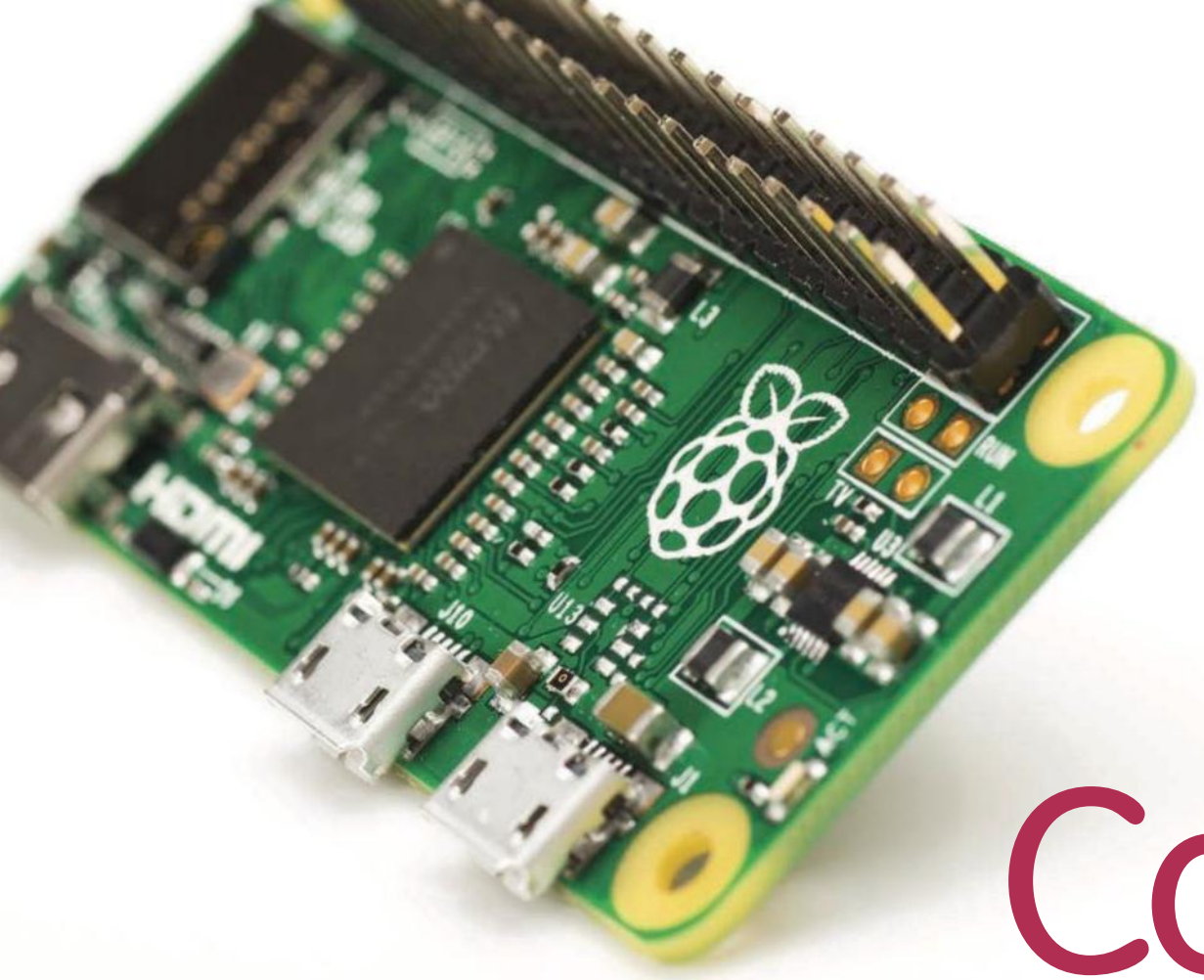


From the makers of

LinuxUser & Developer

Join the conversation at...

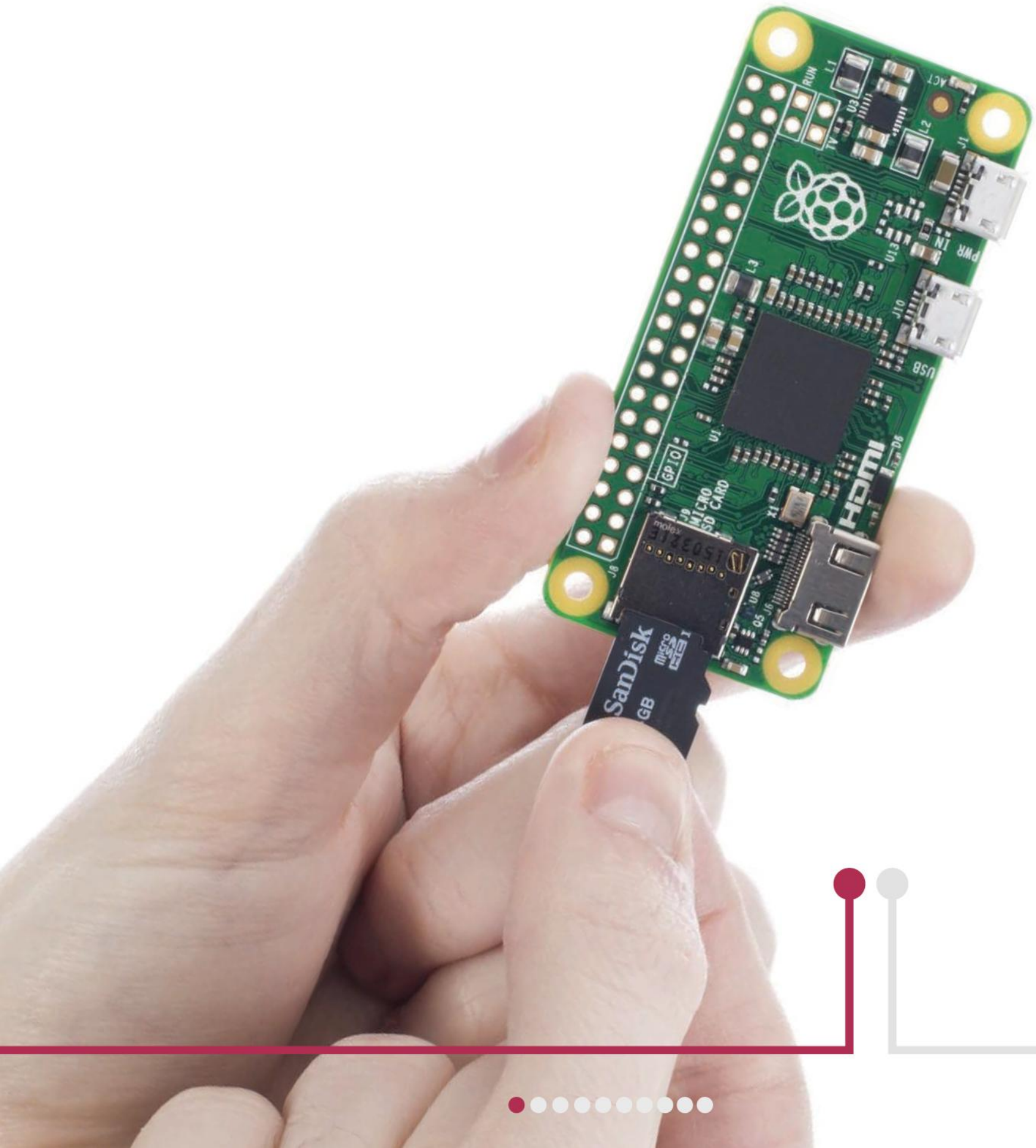
@linuxusermag | Linux User & Developer | RasPi@imagine-publishing.co.uk



Contents

Set up the Pi Zero

Get to grips with your Raspberry Pi Zero, either as a headless device or for use with a screen and keyboard



So you've picked up one of the tiny yet powerful Zeros, but before the coding fun can begin, you need to get more familiar with it. Don't worry; we'll walk you through the Raspberry Pi Zero, the required cables, how to prepare a NOOBS SD card, and how to solder the GPIO header onto the Pi. Once the Pi is working and booted we'll show you how to get it working on Wi-Fi through the Raspbian user interface. You'll need a USB hub for this, or even just to use a keyboard and mouse together. We'll also show you how to prepare a Raspbian SD card for headless use (either VNC or SSH) with only a Wi-Fi adapter or USB-to-Ethernet adaptor.

THE PROJECT ESSENTIALS

Raspberry Pi Zero

microUSB power supply

Soldering iron and solder

Pi Zero adaptor bundle

Monitor, mouse and keyboard (optional)

USB Wi-Fi or USB Ethernet adaptor (optional)

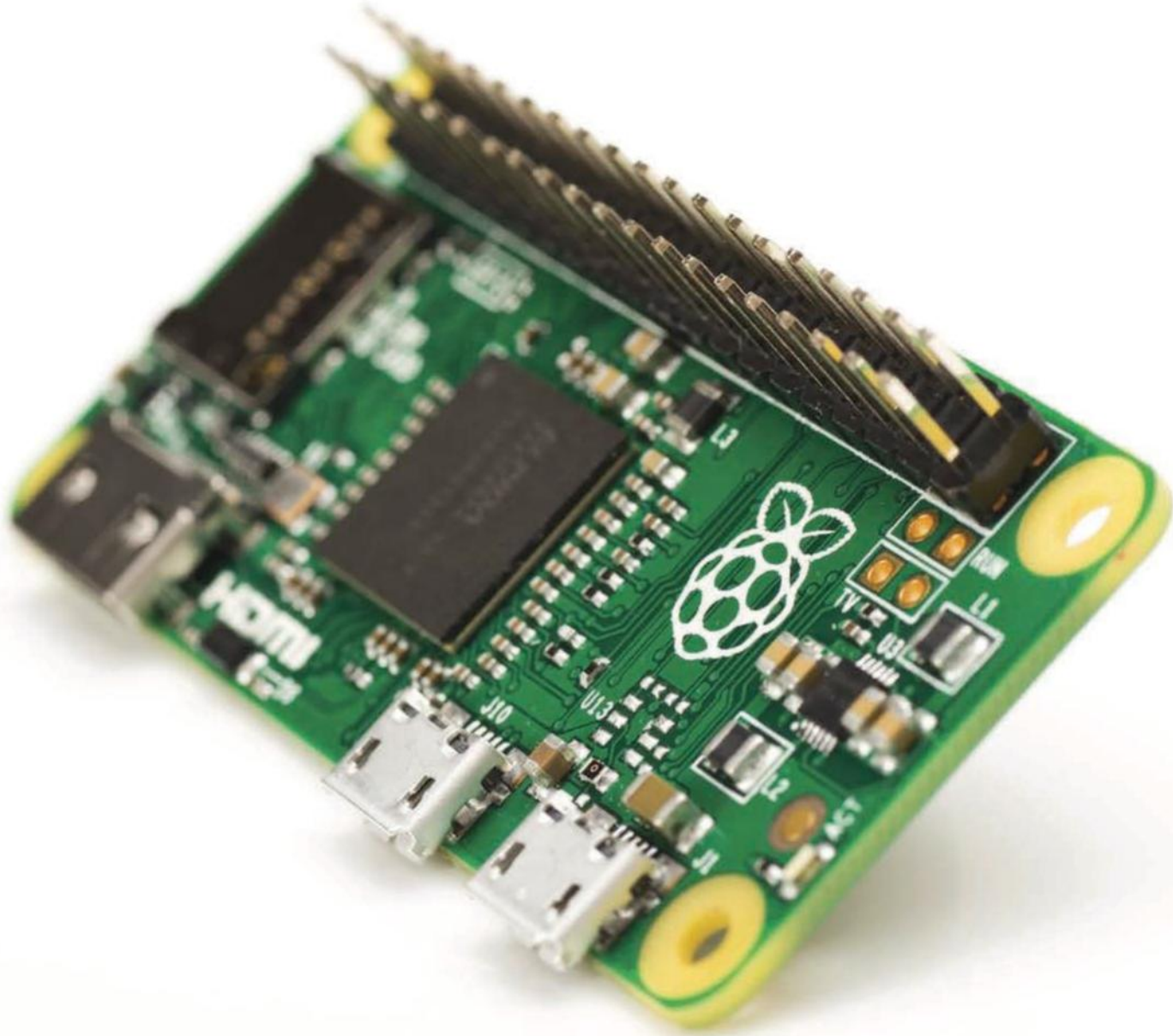
USB hub (optional)

01 Raspberry Pi Zero Cable Overview

The Raspberry Pi Zero is very small, and as such cannot fit normal-sized USB and HDMI connectors on. To use it, you therefore need adaptors that break out microUSB into full-size USB and mini HDMI to full-size HDMI. You also need to be very careful when connecting the microUSB cables as the microUSB power cable will fit into the connector meant for USB data. It's easy to tell them apart though, as they're labelled, and the USB data connector is in between the HDMI and power connectors.

02 GPIO headers

Soldering your brand new Raspberry Pi Zero might seem like a scary prospect at first, but it's not that difficult! What is difficult, however, is snapping off the correct number of GPIO header pins (40), as the kit supplies more than 40. It's also well worth noting at this point that it doesn't matter too much if you mess up and end up missing a couple of the bottom pins!



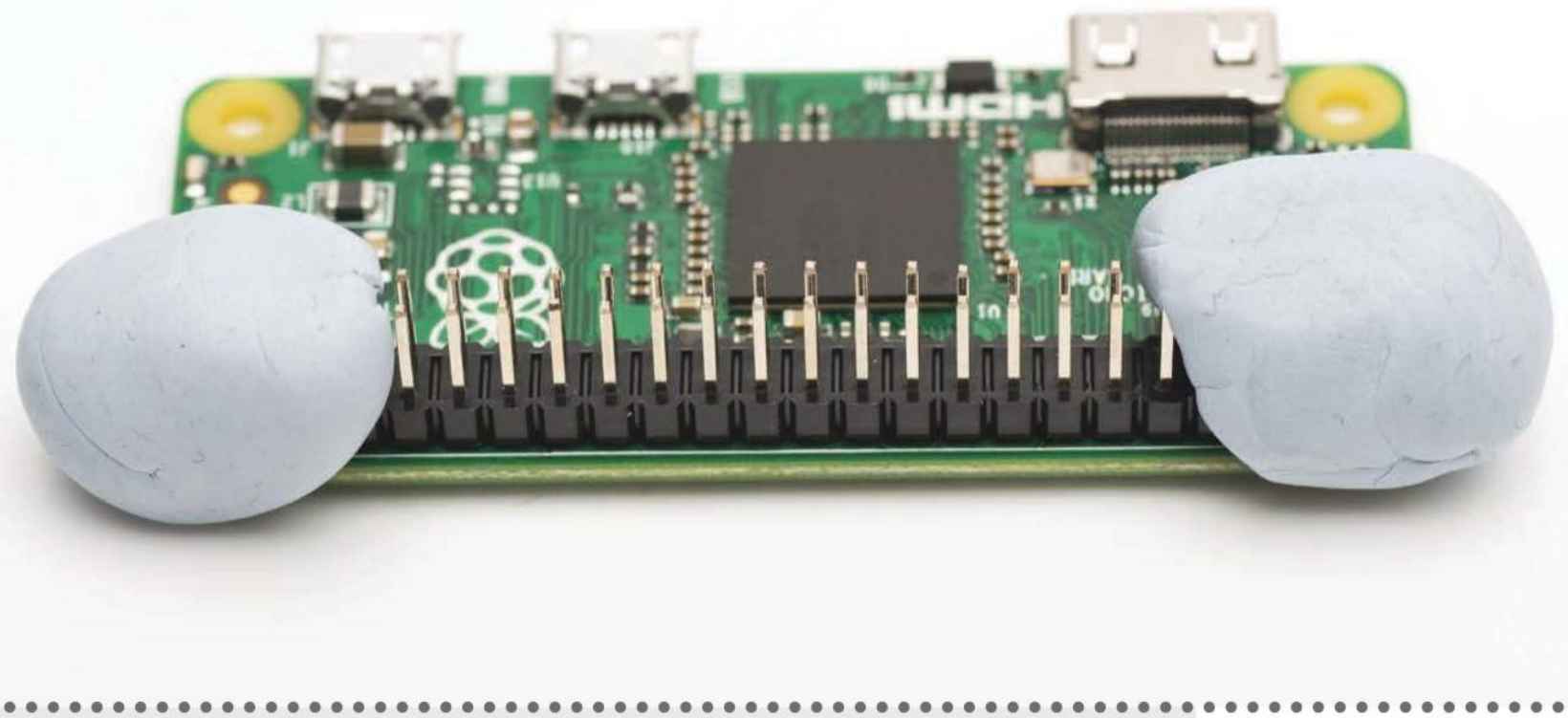
Above Once you've soldered the header into place, your Pi Zero should resemble any other Raspberry Pi

03 Soldering kits

Soldering irons are very cheap these days. If you are going to be doing a lot of soldering then it's probably worth getting a temperature-controlled one where you can change the tip. However, the kit we used with a soldering iron, stand, solder sucker and some lead-free solder was £8 on Amazon. We managed to solder the GPIO pins using this kit no problem.

04 Holding the GPIO headers in place

Before you can solder the GPIO headers, you need to be able to hold them in place. We recommend putting some blu-tack on either side of the pins for this. This

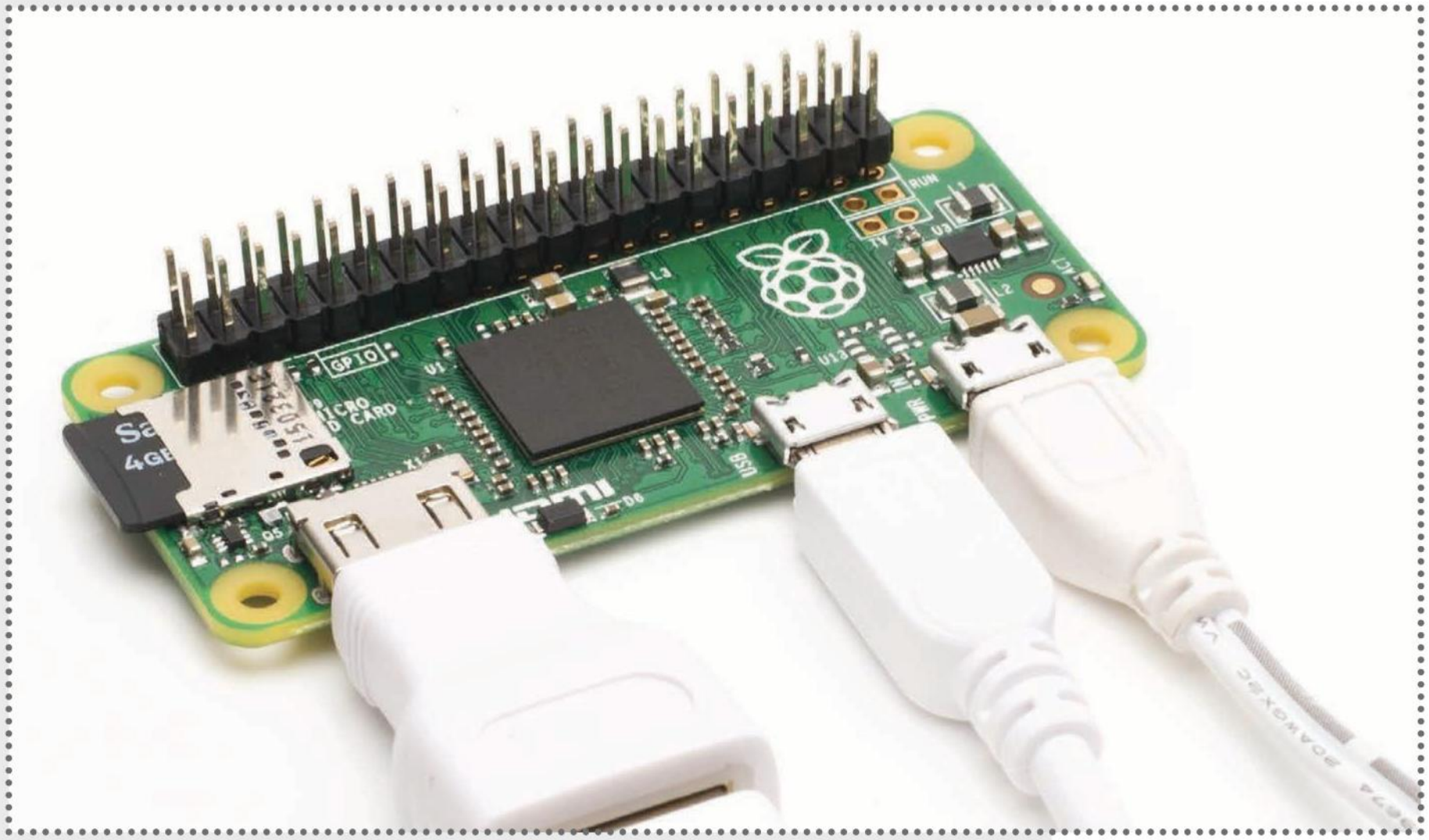


also has the advantage that you can flip the Pi over and then use the blu-tack to keep it in place on a table while you are soldering. The blu-tack should just easily peel off once you are done.

05 Solder the GPIO headers

Here comes the bit you might have been dreading, but don't worry! Make sure you have wet the sponge in the soldering iron holder, as you will need to wipe the iron on the sponge to keep the tip clean. If this is the first time your iron has been used, the heating element will probably give off a lot of smoke for the first few minutes, so don't worry if that happens. Still, be mindful of your safety and make sure that you are soldering in a well-ventilated area – try not to breathe in any fumes.

Once the iron is hot, apply some solder to the tip and wipe any excess solder on the sponge. Then start to solder the pins. For each pin, touch the tip of the iron on the bottom of the GPIO header and the metal contact on the Pi, then apply a very small amount of solder. Once the solder has



flowed onto the pin and the metal contact, then you can remove the iron. If there is too much solder then you can reheat the solder and use the solder sucker to remove it.

Take breaks when soldering the GPIO headers for a couple of reasons: 1) you don't want to overheat any components on the Pi, and 2) you can melt the plastic of the GPIO headers and that will allow the pin to fall through. Keep wiping the tip of the iron on the sponge to keep it clean throughout the soldering process. Make sure you unplug the iron and put it somewhere safe to cool down when you are finished.

06 Prepare NOOBS SD Card

See **www.raspberrypi.org/help/noobs-setup** for more details. NOOBS requires an SD card formatted as FAT32. You then need to download the latest NOOBS image from **https://downloads.raspberrypi.org/NOOBS_latest** and then unzip it to the SD card. On Linux, the steps are as follows:

```
sudo parted /dev/mmcblk0
(parted) mktable msdos
(parted) mkpart primary fat32 0% 100%
(parted) quit
sudo mkfs.vfat /dev/mmcblk0p1
cd /mnt
sudo mkdir pi
sudo mount /dev/mmcblk0p1 pi
cd pi
sudo unzip ~/Downloads/NOOBS_v1_5_0.zip
sync
cd ..
sudo umount pi
```

"If you are going to be doing a lot of soldering then it's probably worth getting a temperature-controlled soldering iron"

07 Boot NOOBS and install Raspbian

Connect your Pi Zero up as shown in the first step. The minimum you need connected for a NOOBS install is a monitor and a keyboard. However, a mouse and either an Ethernet adaptor or Wi-Fi adaptor are also very useful. Press Enter to select Raspbian and then press I to install. Then press Enter to agree. Once it is finished it will say 'OS installed successfully'. Press OK and your Pi will reboot into Raspbian. Alternatively, if you don't want to use NOOBS, you can flash Raspbian to an SD card in the usual manner. Raspbian will boot into a desktop environment by default.

08 Configure Wi-Fi

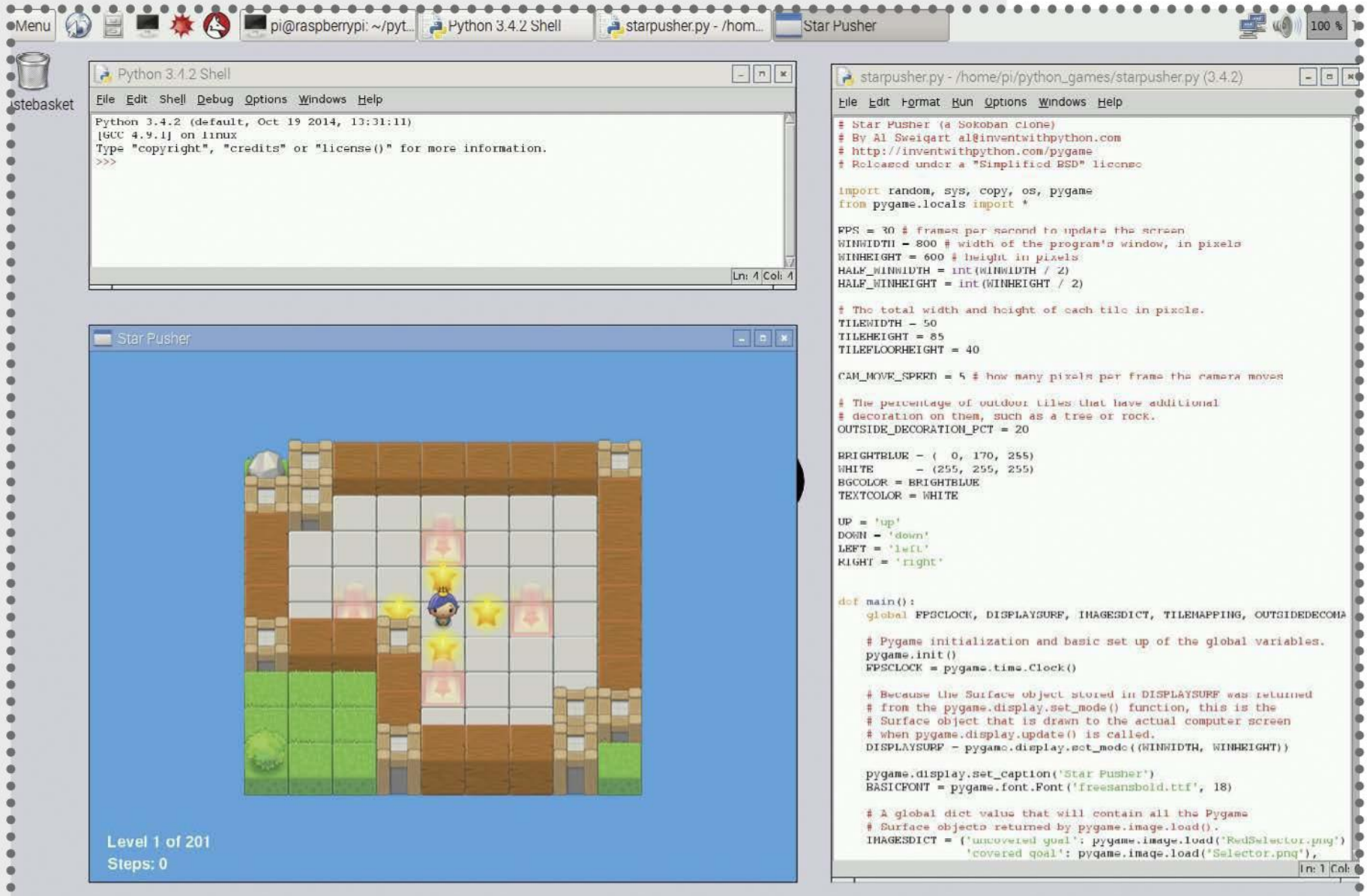
If you are using a USB-to-Ethernet adaptor then the Pi should already be connected to the internet. If you are using a Wi-Fi adapter then you will need to configure it to connect to your wireless network. We are using an Edimax EW-7811UN, which works perfectly with the Pi out of the box.

Once at the Raspbian desktop, you can click on the network icon in order to see the available wireless networks. Once you click on one it will ask you for the password. After that it should be associated; you can hover your mouse over the icon and see the networks that you are connected to.

09 Configure Wi-Fi from another machine

If you want to use the Pi Zero as a headless device with Wi-Fi then you can prepare an SD card using another Linux machine that will already be configured to connect to the correct Wi-Fi network. You have to mount the SD card and edit /etc/wpa_supplicant/wpa_supplicant.conf, which is the same file that is configured by the Raspbian user interface from the previous step. Insert the SD card into your Linux machine and work out what the device is called.

```
dmesg | tail -n 3
[320516.612984] mmc0: new high speed SDHC card at address 0001
[320516.613437] mmcblk0: mmc0:0001 SD8GB 7.35 GiB
```



So the device is /dev/mmcblk0 – now we need to work out which partition number the root partition is (this will be different on a Raspbian image; we are using a NOOBS image here).

```
sudo parted /dev/mmcblk0 print
```

This will give you a list of the partitions. The largest partition will be the root partition. In this case it's partition 7, so the root filesystem is at /dev/mmcblk0p7. To mount the SD card and edit the wpa_supplicant.conf file, do the following.

```
cd /mnt
sudo mkdir pi
sudo mount /dev/mmcblk0p7 pi/
```

Above The Zero may be tiny but it is just as good for programming

```
cd pi/
sudo nano etc/wpa_supplicant/wpa_supplicant.conf
```

Then fill in your Wi-Fi details:

```
network={
   ssid="your_wifi_network"
   psk="your_wifi_password"
   key_mgmt=WPA-PSK
}
```

Then finally:

```
cd ..
sudo umount pi/
```

10 Remotely access your Pi

You can use nmap to scan the local network to find a Raspberry Pi. You need to know the address range of your local network (common networks are 192.168.1.0/24, and 192.168.2.0/24). You can find it with the ip addr command. nmap -p22 -sV 192.168.157.0/24 will scan for a list of devices with SSH open. Example output:

```
Nmap scan report for 192.168.157.29
Host is up (0.070s latency).
PORT   STATE SERVICE VERSION
22/tcp open  ssh     (protocol 2.0)
```

Then you can SSH in with:

```
ssh  pi@192.168.157.29
```

Add Wi-Fi capability

If soldering the GPIO headers went well for you, you could take it to the next level and attempt to solder the internals of a USB Wi-Fi adapter straight onto the Pi Zero. This is useful if you really need to save space and are using the Pi Zero as an Internet of Things device. See **http://hackaday.com/2015/11/28/first-raspberry-pi-zero-hack-piggy-back-wifi** for more details on this.

The password is 'raspberry'. If you are using the Pi headless, you'll want to disable the user interface that is started on boot by default:

```
sudo systemctl set-default multi-user.target
```

11 Set up a VNC server

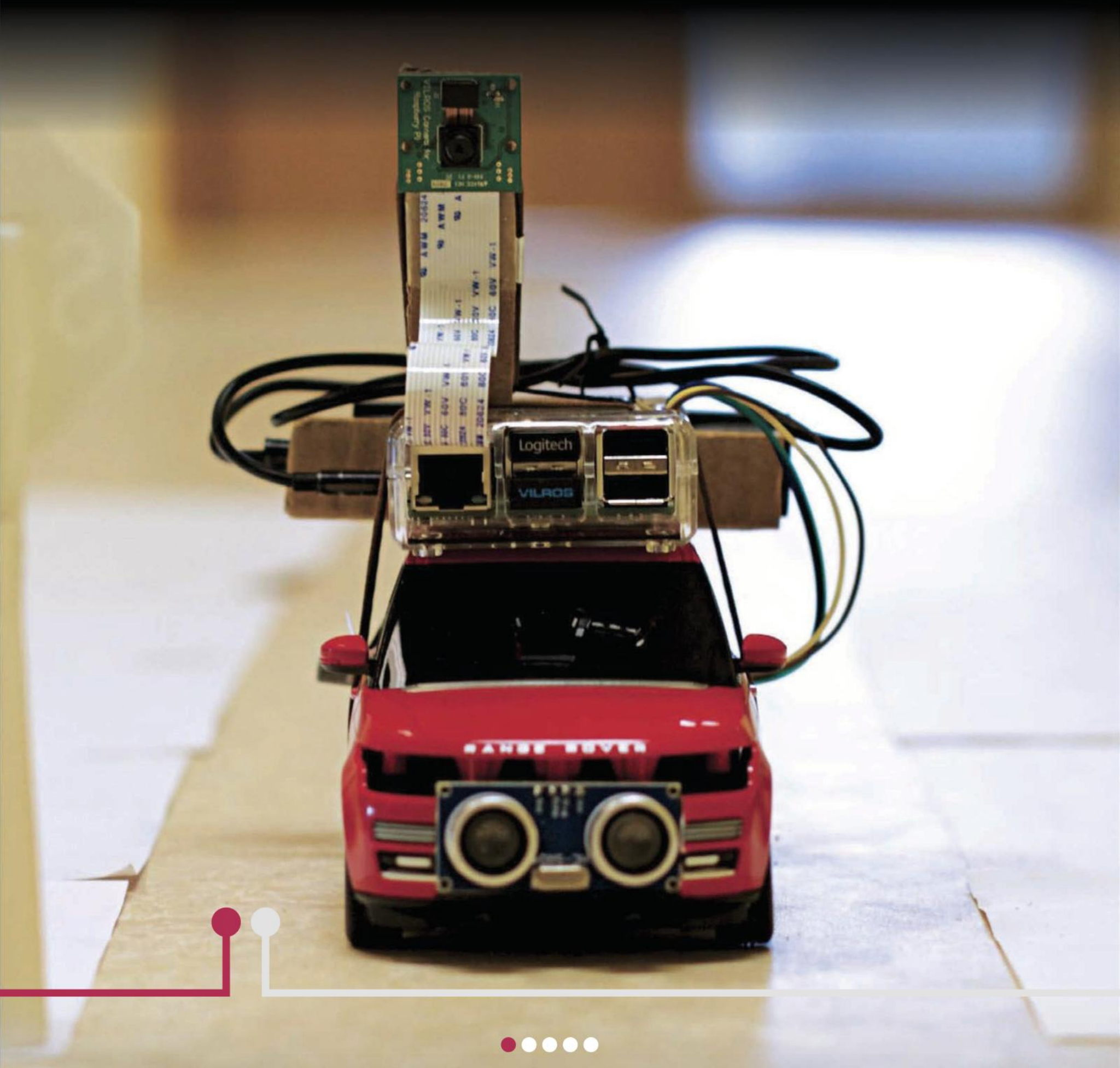
VNC stands for Virtual Network Computing. Using VNC you can access the Raspbian desktop over the network (meaning you only need power and Ethernet/Wi-Fi connected). There is no audio support, but for any other tasks (including the use of pygame) VNC should provide acceptable performance. You can install a VNC server with the following commands:

```
sudo apt-get update
sudo apt-get install tightvncserver
```

There are several free VNC clients available so a search engine will help you find a suitable one. To start a VNC session on your Pi, log in over SSH and then run tightvncserver. You will be prompted to enter a password the first time you run it. You can specify a screen resolution with the -geometry option: for example, -geometry 1024x768. You can kill an existing vnc session with tightvncserver -kill :1, where 1 is the session number. To connect to that session on a Linux machine, you could use the command: vncviewer 192.168.157.29:1, substituting for the IP address of your Raspberry Pi.

Self-driving RC car

Zheng Wang turns the tables on Google with his very own fully-functioning self-driving car



Where did the idea to develop a self-driving car come from?

Believe it or not, I actually did this for a school project. A lot of my interests centre around machine learning, so I decided to do something that heavily involves machine learning and the concepts that surround it. I did some research online and found a very inspiring self-driving car project made by David Singleton, which showcased what he was able to achieve with just an Arduino board and a few other items. I was amazed to see that the RC car can drive itself along the track without aid and wondered if I could replicate a similar project.

After that, I took out my Raspberry Pi and made up my mind to attempt to build my own self-driving RC car that could do even more. The aim was to include things like front collision avoidance, stop sign and traffic light detection. It took me a while to develop the project to anything more than an idea, just because there are so many factors that needed to be considered.

Could you give us an overview of how the self-driving system works?

The crux of the system consists of three subsystems that work seamlessly in sync together. These systems consist of an input unit for controlling the camera and ultrasonic sensor, a processing unit and also the main RC car control unit.

Firstly, live video and ultrasonic sensor data are streamed directly from the Raspberry Pi to the computer via a strong Wi-Fi connection. I was quick to recognise that it was imperative to create as little latency as possible in the streaming, so in order to achieve these goals, the video resolution is dramatically scaled down to QVGA (320×240). It provides that smooth streaming experience that I was after. The next step is for the colour images received on the

Zheng Wang has an academic background in electrical engineering and has put what he has learned into action with his Pi car project.

computer to be converted into greyscale and then fed into a pertained neural network to make predictions for the car; so whether it should go straight ahead, or make a left or right turn at the correct moment. These same images are used to calculate the stopping distance between the car and the stop signs, while the Raspberry Pi alerts the system of the distance to an upcoming obstacle. The object detection in this project is primarily learning based.

The final part of the system consists of outputs from the artificial neural network that are sent to the Arduino via USB, which is connected directly to the RC controller. The Arduino reads the commands and writes out LOW or HIGH signals, simulating button-press actions to drive the RC car. With so many sensors and data feeds consistently taking place, there was a lot of initial trial and error involved, but it didn't take me an overly long period of time to get the project running completely independently.

What sort of role did the Raspberry Pi play in the grand scheme of things for your self-driving car?

The main benefit of using the Raspberry Pi was that it's the perfect piece of apparatus to help collect input data, which is a massive part of this project. With the Raspberry Pi in place, I connected a Pi camera module and an ultrasonic

THE PROJECT ESSENTIALS

Raspberry Pi B+

Arduino

Camera module

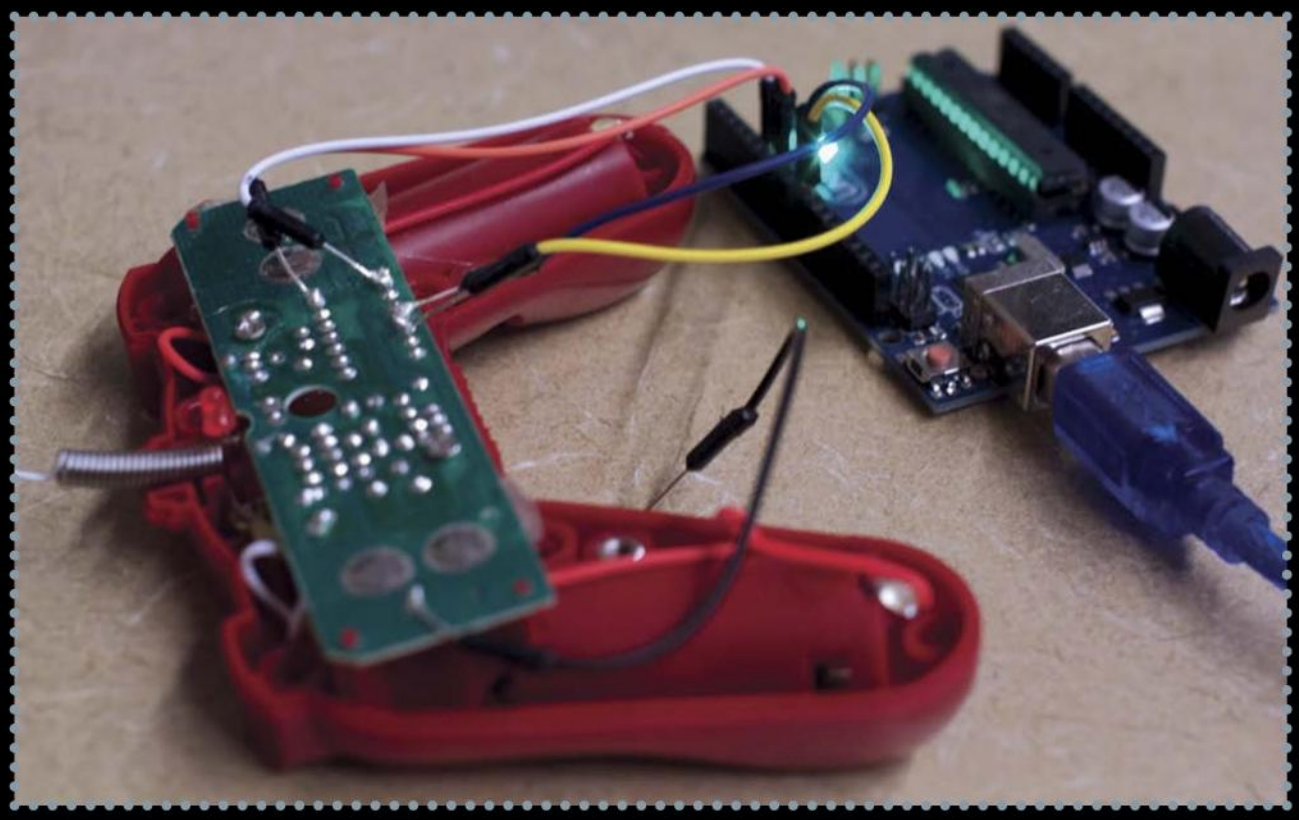
HC-SR04 ultrasonic sensor

OpenCV

sensor, which work in tandem to help the Pi collate its data. There are also two client programs running on the Raspberry Pi that help with the streaming side of things. One is solely for video streaming and the other is for the data streaming from the ultrasonic sensor. To be honest, I didn't stray too far from the official picamera documentation when using it, as all the guidelines for video streaming are all in there. When I needed some help with measuring distance with the ultrasonic sensor, there were some handy tutorials on the web for fellow enthusiasts to follow and there's other reference material all over the place.

Can you tell us more about the ultrasonic sensor? Can it detect collisions at a full 360 degrees?

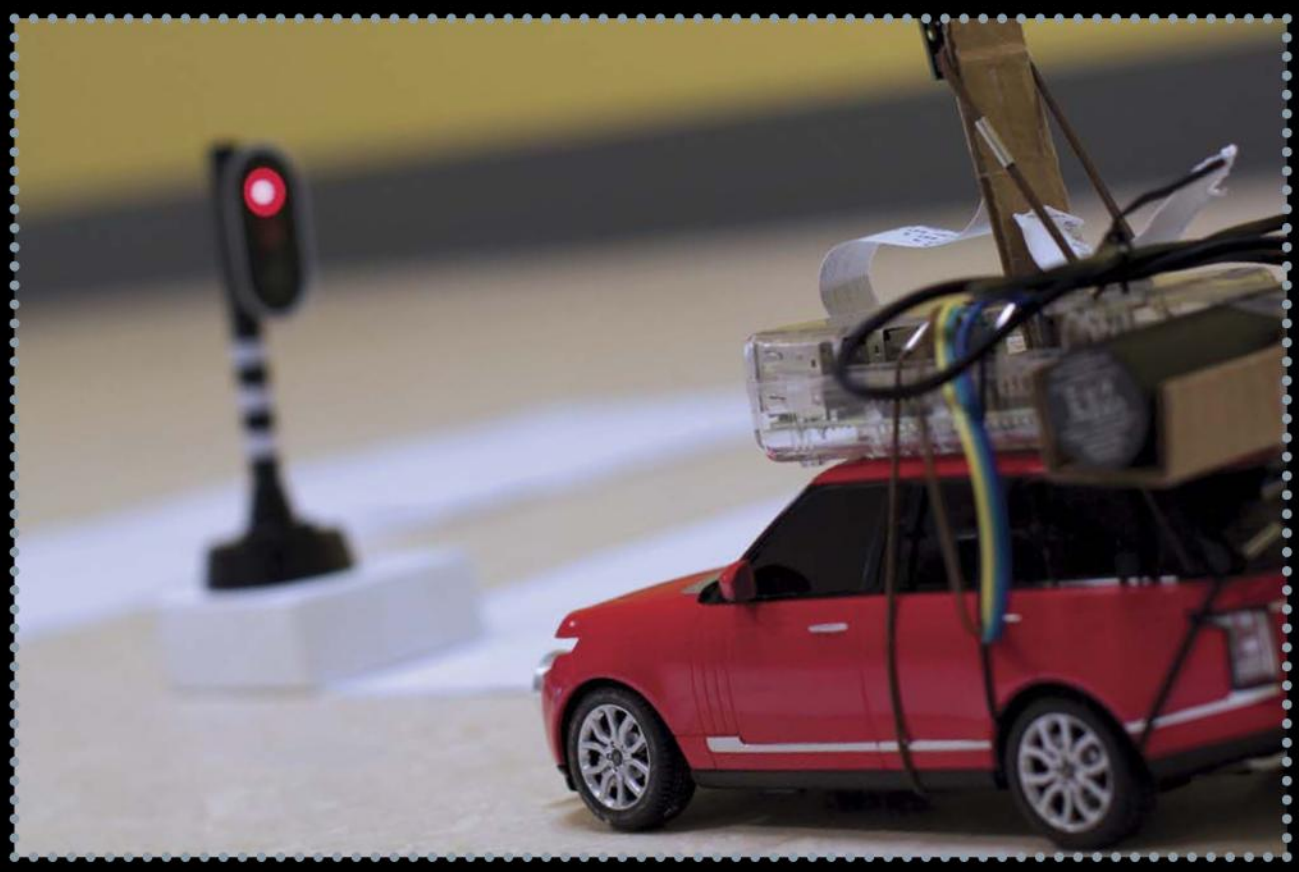
For this project, I chose to use the HC-S404 ultrasonic sensor, as it's one of the most cost-effective and user-friendly pieces of kit on the market. It can be a bit fiddly to set up from scratch, but as I mentioned previously, I was able to source help from the internet whenever I had a problem that I needed solving. For this sensor in particular, the manual lists its best detection is within 30 degrees,



which would seem about right based on the tests that I have run with it. There are numerous sensors on the market, so a complete 360-degree detection seems like something that would be plausible.

How do you see yourself taking this project further? Perhaps you'll want to scale up to a bigger model?

There are a lot of areas that I'd like to explore further to really take my self-driving car to the next level. For one, I'd like to eliminate the use of the ultrasonic sensor and instead implement a stereo camera for measuring the distances. The results are far more accurate than what the ultrasonic sensor can offer. If I get into the situation where I've got more spare time on my hands, perhaps I'll look to add new behavioural features. It would be intriguing to see if I can implement things like lane changing and overtaking into the project. Outside of this project, I'm not working on any other Raspberry Pi projects currently, but I'm always on the hunt for new inspiration – and the Pi is an amazing piece of kit that I love to work with.



Make an interactive, Zero-powered wearable

Harness the Twitter API to trigger displays in an LED-laden piece of clothing when set tweets are received

Wearable tech is an ever-growing industry, bursting with smart watches, fitness gadgets and pulse-rate necklaces. For many, this technology is nothing new; enthusiasts have long created their own wearable versions. Clothes that light up on contact, masks that change voices and weapons that glow! In this tutorial you will use your old Christmas LED lights, Python and the Pi Zero to modify a hat that lights up when you receive a specific tweet from your timeline. The Pi Zero is the perfect size and can be embedded into the clothing. You can customise the project for your own wearable tech, perhaps shoes that light up or a pair of gloves or a jumper that responds to the weather.

THE PROJECT ESSENTIALS

Pi Zero

USB portable power supply

USB Wi-Fi dongle

Micro USB convertor

Hat or other clothing of your choice

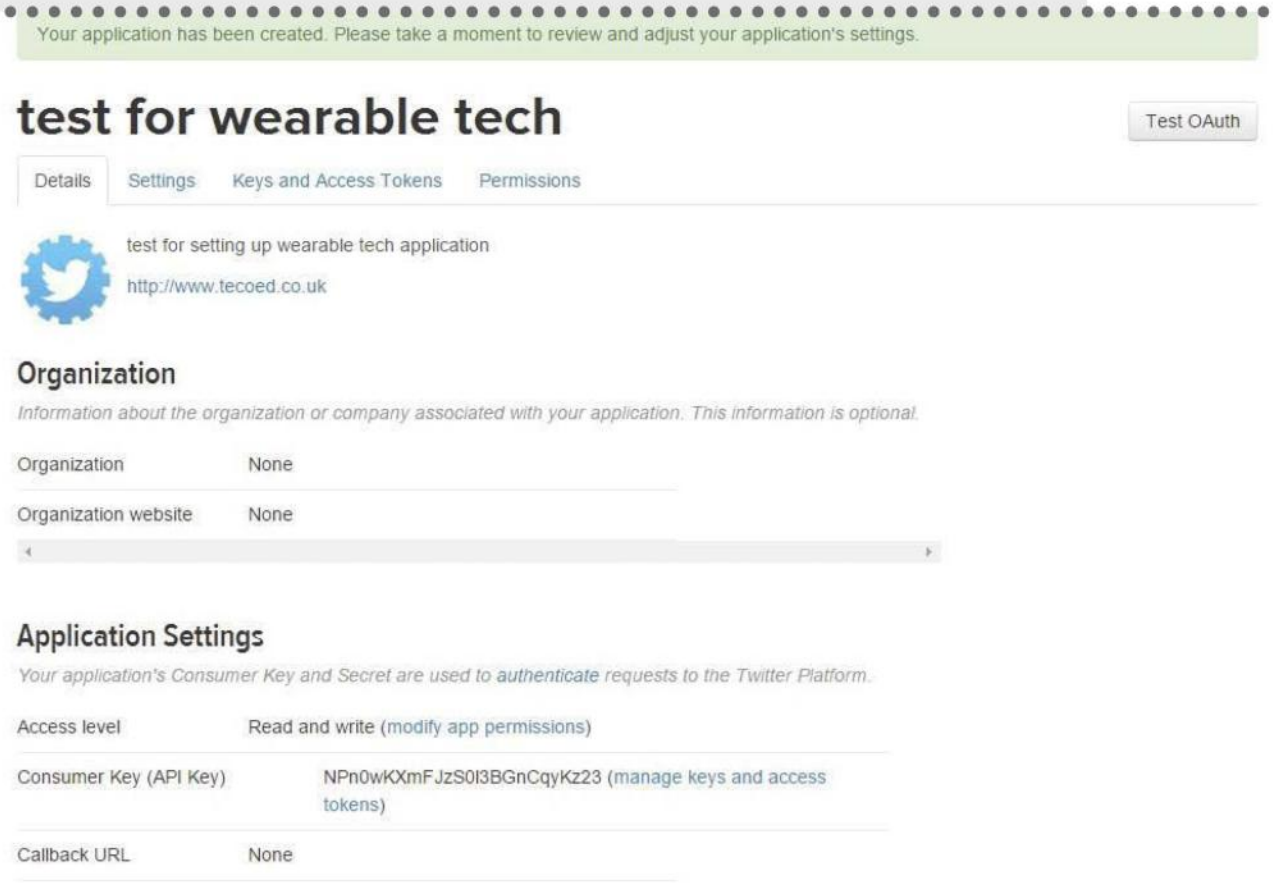
Old LED Christmas lights

01 Sign up for Twitter API keys

To interact with Twitter, you first require consumer and authorisation keys which are available from the Twitter API site. If you already have these, you can jump to Step 04, but if not we'll take you through it all here. Head over to **https://apps.twitter.com** and sign in with your regular Twitter username and password. Select the 'create a new app' button. Fill in the details for your app as required, then tick and confirm that you accept the terms and conditions.

02 Permission settings

On completion of this, you will be taken to the app overview page. Here you need to select the 'Access Level Permission' for your app. (This may require you to register a mobile phone number, which is completed in your regular Twitter account settings page.) There are three permission settings:



Above We'll need to register our application with Twitter in order to use it in our hat

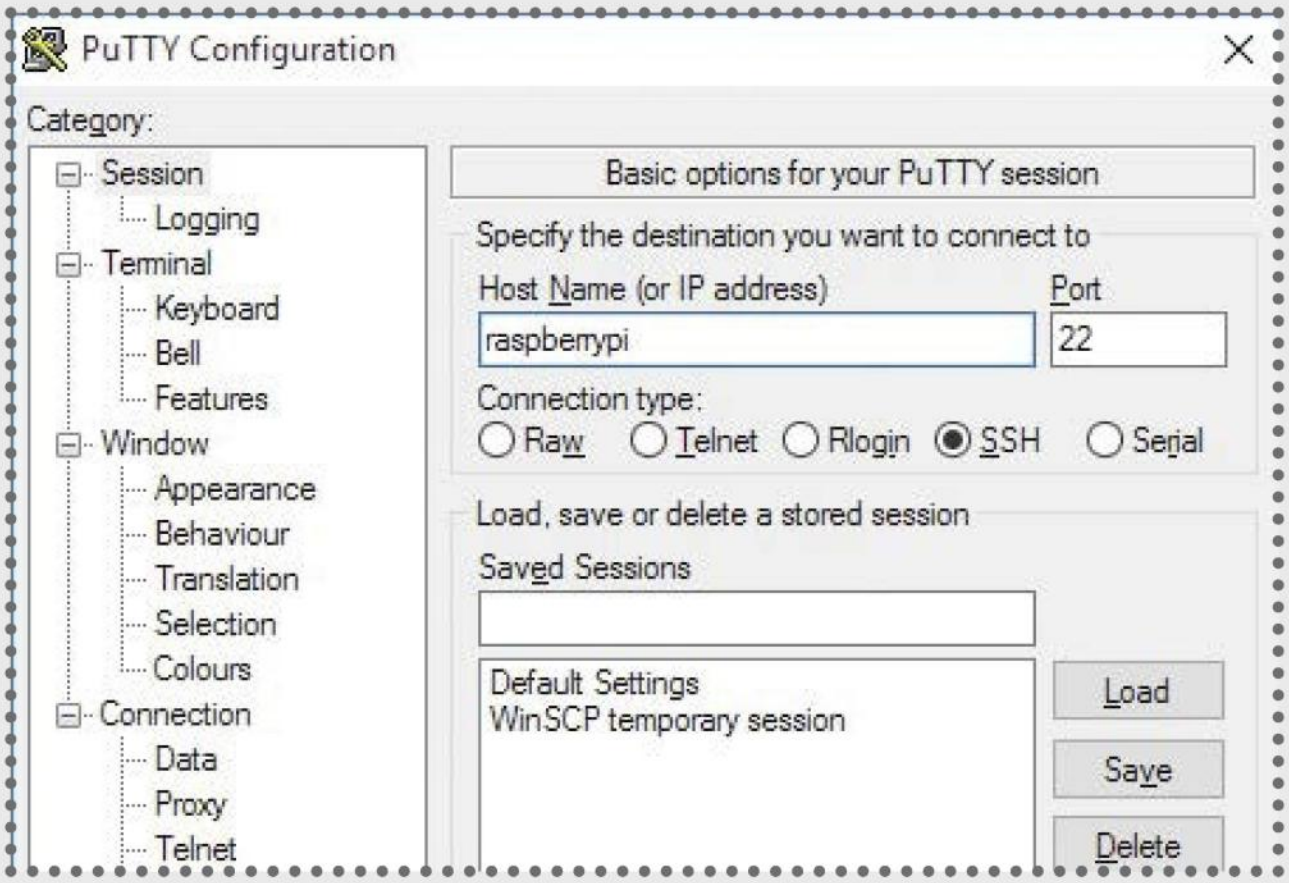
Read: This reads tweets from your timeline.
Read and write: This enables you to read tweets and write/send tweets back to your timeline.
Read, write, direct: This permission setting permits you to send and access your direct messages from your timeline.

For these sorts of projects, you will need to set the permission to 'Read and write'.

03 API keys

Now set up your 'keys'. These are the codes that are used to access your Twitter account via Python. Click the 'Keys and access token' page. You will be presented with your consumer key and consumer secret. Copy these down as they are required in Step 7.

At the bottom of the page is the 'access tokens' that are generated. Simply press the button and this



will generate them. Again, note these down for use in Step 07. Each time you press the button, a new set of keys will be created; this is useful if they ever become compromised.

04 Set up the Wi-Fi

The Pi Zero is the perfect size for embedding into projects and hacks. When using it within clothing it is unlikely that you will have a display or screen. A simple method to access your Pi is via SSH (secure shell) – this requires Wi-Fi access. An easy way to set this up is to hook up the Pi Zero to a monitor, boot it up, select the Wi-Fi icon in the top-right corner and enter in your pre-shared WEP key. Save the settings and each time your Pi Zero starts up, it will attempt to connect to this Network.

05 SSH into the Pi Zero

Now the Wi-Fi is operational, download and install an SSH client such as Putty onto your laptop. Run Putty, enter the IP address of your Pi Zero (usually you can

use the default name raspberrypi). Enter your user name and password when prompted and you will be presented with the terminal window. This can be used to write code, run code and set up the project.

06 Install Tweepy

You have now registered your Twitter app, acquired the consumer keys and tokens, and have access to your Pi. The next step is to download Tweepy, the Python Twitter API. In the LX Terminal type:

```
sudo apt-get install python-setuptools
sudo easy_install tweepy
```

... or...

```
sudo pip install tweepy
```

Reboot your Pi Zero typing, sudo reboot. You will then need to SSH in again. You can now use Python code to interact with your Twitter feed.

07 Connect to Twitter

To stream your tweets, you need to authorise a connection using your consumer keys and the access token that you set up in Step 03. Create a new Python file – this can be completed at the command line by typing sudo nano name_of_file.py. Add the lines of code below, which will stream tweets down to your Pi Zero. Line nine listens for tweets to your timeline and then tweet_to_check = tweet.text grabs each tweet. Use print tweet_to_check to print the tweet. Save the

Tethering the Zero

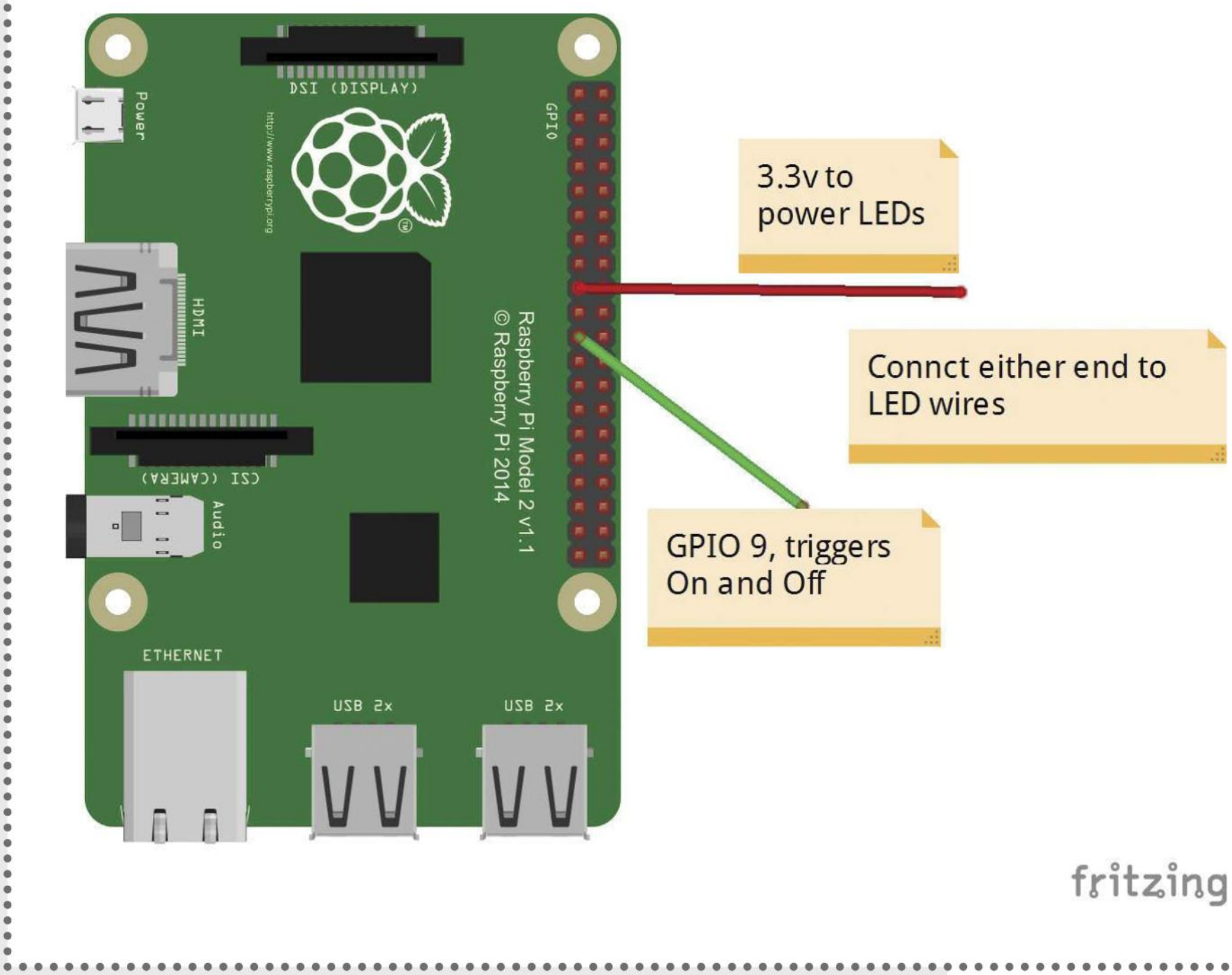
The Pi Zero can be tethered to a mobile phone to ensure that the Hat is mobile-interactive when out and about. To maintain the connection, it is advisable to disable the Wi-Fi management. To set this up type: sudo nano /etc/network/interfaces Then add the line: wireless-power off. Save the file and reboot the Pi.

file using Control + X, then to run the program type sudo python name_of_file.py. Test that it is working before moving onto Step 08. The program will stream each of the tweets in your timeline and print them out.

```
import sys, subprocess, urllib, time,
tweepy

consumer_key= 'xxxxxxxxxxxxx'
consumer_secret= 'xxxxxxxxxxxxxxxxx'

access_token= 'xxxxxxxxxxxxxxx'
access_token_secret= 'xxxxxxxxxxxxxxxxx'

auth = tweepy.OAuthHandler(consumer_key,
consumer_secret)
auth.set_access_token(access_token, access_
token_secret)

api = tweepy.API(auth)

class Hat_Lights(tweepy.StreamListener):
    def on_status(self, tweet):

        tweet_to_check = tweet.text ##gets
the tweet
        print tweet_to_check

stream = tweepy.Stream(auth, Hat_Lights())

while True:
    stream.userstream()
```

Add Wi-Fi without a display

You can set up the Wi-Fi connection via the SD card before you boot up your Pi. This involves a few more steps than the GUI interface but will give you some understand of the different elements of information required to connect to the Wi-Fi. Check out this forum guide from the Raspberry Pi website: **https://www.raspberrypi.org/forums/viewtopic.php?f=26&t=34127.**



08 Prepare the LEDs

There are two ways to power the LEDs depending on the item of clothing you are modifying. For example, if using a hat, you may want to avoid a battery pack as it will add extra weight. Take one of the wires and cut it in half between the LED and the battery pack. Solder to each end a male-to-female jumper wire.

The Pi Zero can provide 3.3V or 5V – attach one of the wires to provide power to the LEDs via, for example, GPIO pin 2, 4 or 9, and then the other wire to another pin, such as pin 11. Or use a non-power pin and a ground pin, with the power for the LEDs being

Above Here's the simple Fritzing diagram for the hat connections

provided by the batteries. It is not advisable to use both as this will quickly burn out the LEDs.

09 Test the LEDs

Next, check that the LED and wiring are working by using the simple test program below. Create a new Python file (sudo nano led_test.py) and add the code below. Save the file and type sudo python led_test.py to test the LEDs. They will turn on for ten seconds and then off again. Replace the GPIO pin numbers below with the ones you are using:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(9, GPIO.OUT)

###Turn the lights on
GPIO.output(9, GPIO.LOW)
time.sleep(10)
###Turn them off
GPIO.setup(9, GPIO.HIGH)
```

10 Add the modules

Now you have both Twitter and your LEDs ready to go, return back to the Twitter test program you created in Step 7. At the top of the program, import the required modules to enable the Python program to interact with Twitter and control the LEDs (lines 3 to 6). The final line adds a 15-second delay to the start of the program. This is to ensure that the Pi Zero has time to connect to the Wi-Fi before it connects to Twitter, otherwise you can receive an authorisation error.

"To stream your tweets, you need to authorise a connection"

```
import os
import time;
import sys, subprocess, urllib, time,
tweepy
import RPi.GPIO as GPIO

time.sleep(15)
```

11 Check for a key word

To ensure that a user actually meant to turn on the LEDs, the program looks for an exact tweet. For example, @ada_lovelace ON. In your program, add the line tweet_to_check.find("@Your_Twitter_ID ON") (line 31) which will return the numerical position of the phrase. This will always be zero as the phrase starts from position zero. If the value is not zero then the exact phrase has not been sent. A simple if statement can be used to respond to the value (line 34):

```
does_the_tweet_contain_key_word = tweet_
to_check.find("@Test_User ON")
```

12 Get user name

Once the program has checked that a Twitter user has tweeted the 'phrase', the next stage is to retrieve the user's Twitter ID (line 38). This enables you to message them a templated confirmation that they have triggered the LEDs. Use the line user = str(tweet.user.screen_name) to add the ID to a variable called user.

13 Find the time

If you send the same tweet multiple times, Twitter assumes that you are a spam bot and will not permit the message to be sent. To get around this, record the time that the user triggered the LED lights and add the time to the message (line 39). This also makes the tweets appear in real-time. To retrieve the time that the interaction occurred, use:

```
time_sent = time.asctime( time.
localtime(time.time()) )
```

This will check the local time of your Raspberry Pi and save it to the variable 'time_sent'. (Ensure that your Pi's clock is set correctly before you start your program.)

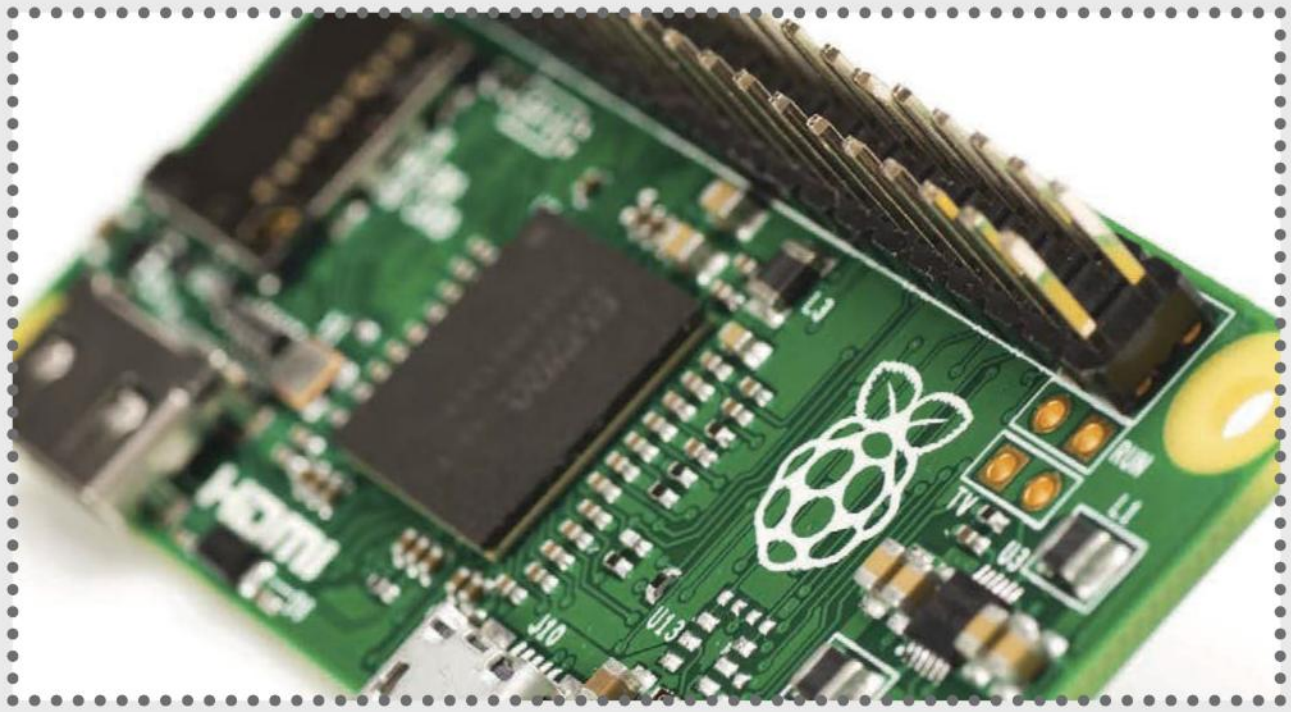
14 Add a picture and message to the tweet

This step combines your reply, a picture and the 'time' from the previous step to create a final message which is sent to the Twitter user who triggered your lights.

Create a new variable called message and add the text and the time: message = "You turned on the Hat!", time_sent (line 40). Locate a suitable picture that you wish to send and also store this as a variable: pic = '/home/pi/lights.jpg' (line 37).

15 Combine the parts of the message

Combine the two parts from the previous steps to create your final message, which is stored in a variable called final_tweet using this code: final_tweet = "@%s" %(user), message (line 42). The "@%s" %(user) adds the user's Twitter ID to the message, which



ensures that your tweet will appear in their timeline. Finally, send your tweet using the update code:

```
api.update_with_media(pic, final_tweet)
```

16 Turn the lights on

This is same section of the code used in Step 09, which switches the GPIO pin to low, creating a circuit and allowing the current to flow through. This turns on the LED lights. Add a time delay (line 46), so that they stay on for a short period of time before breaking the circuit and then go off again: GPIO.setup(9, GPIO.HIGH) (line 48).

```
###Turn the lights on
GPIO.output(9, GPIO.LOW)
time.sleep(10)
###Turn them off
GPIO.setup(9, GPIO.HIGH)
```

17 Automatically start the program

Since the program is running on a Pi Zero and embedded into the hat, it needs to start automatically

when the power is turned on. To enable the program to run at startup, type this into the LX Terminal: sudo nano /etc/rc.local. At the bottom of the file, add the file path of the program – for example, python /home/pi/LED_Hat.py &. Don't forget the & symbol at the end! Reboot the Pi Zero and the program will start automatically. Pop your hat on and get your followers to tweet at you!

Build a Pi cluster with Docker Swarm

Combine the power and resources of your Raspberry Pis by building a Swarm with Docker

Docker is a framework and toolchain used to configure, build and deploy containers on Linux. Containers provide a means to package up an application and all its dependencies into a single unit. This makes them easy to share and ship anywhere, giving a lightweight and repeatable environment.

Each application runs in its own isolated space sharing the host's kernel and resources, in contrast to a virtual machine which needs to ship with a full operating system. A Docker container can be started or stopped within a second, and can scale to large numbers while having minimum overhead on the host's resources.

The Docker community has built out a clustering solution called Swarm which, as of version 1.0, is claimed to be "production ready". Our single Raspberry Pi has 1GB RAM and four cores, but given five boards we have 20 cores and 5GB RAM available. Swarm can help us distribute our load across them.

Get ready to install Arch Linux, compile Docker 1.9.1 from source, build some images and then start up your own swarm for the first time.

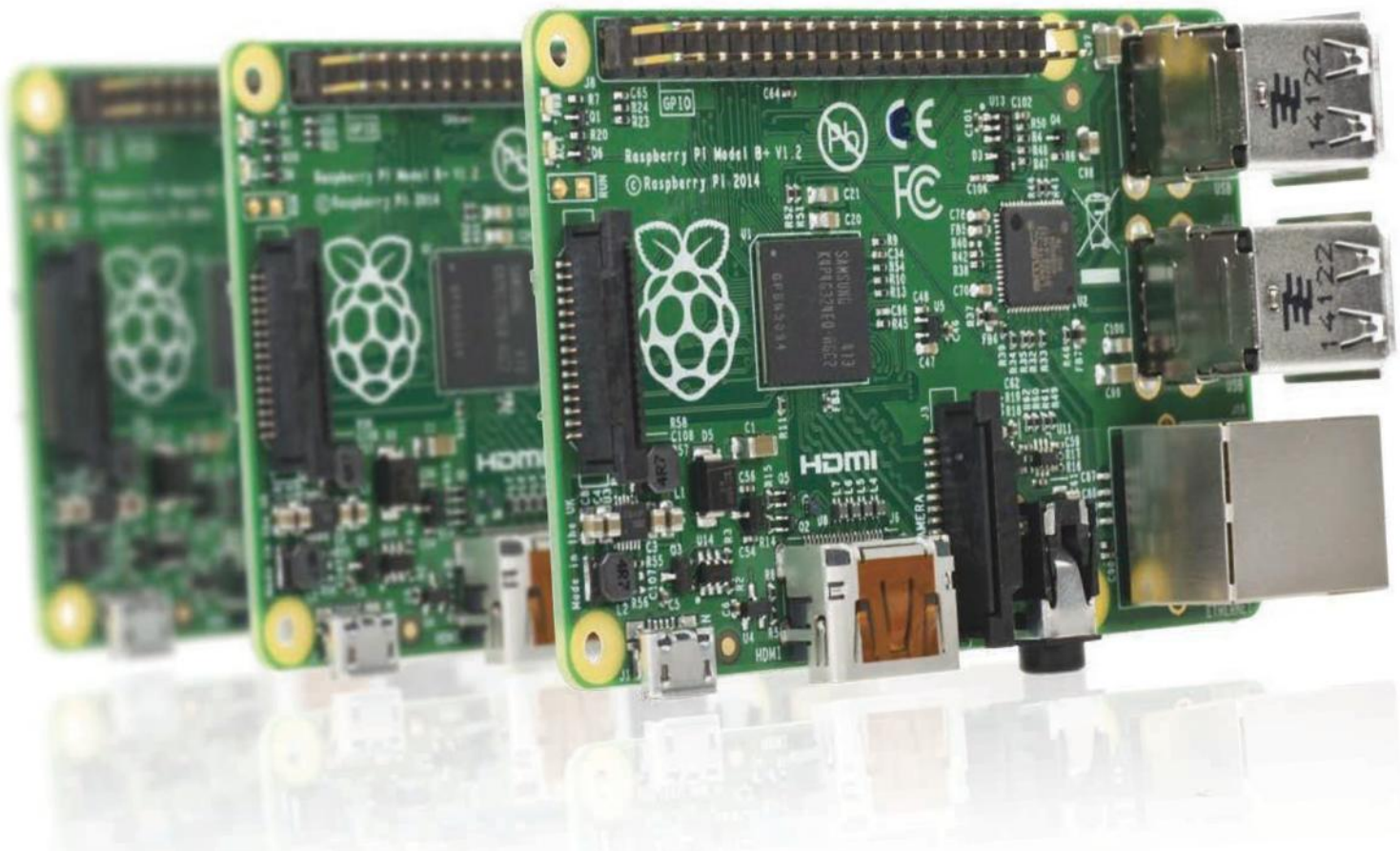
THE PROJECT ESSENTIALS

Github repository

https://github.com/alexellis/docker-arm

Arch Linux for ARM

https://archlinuxarm.org

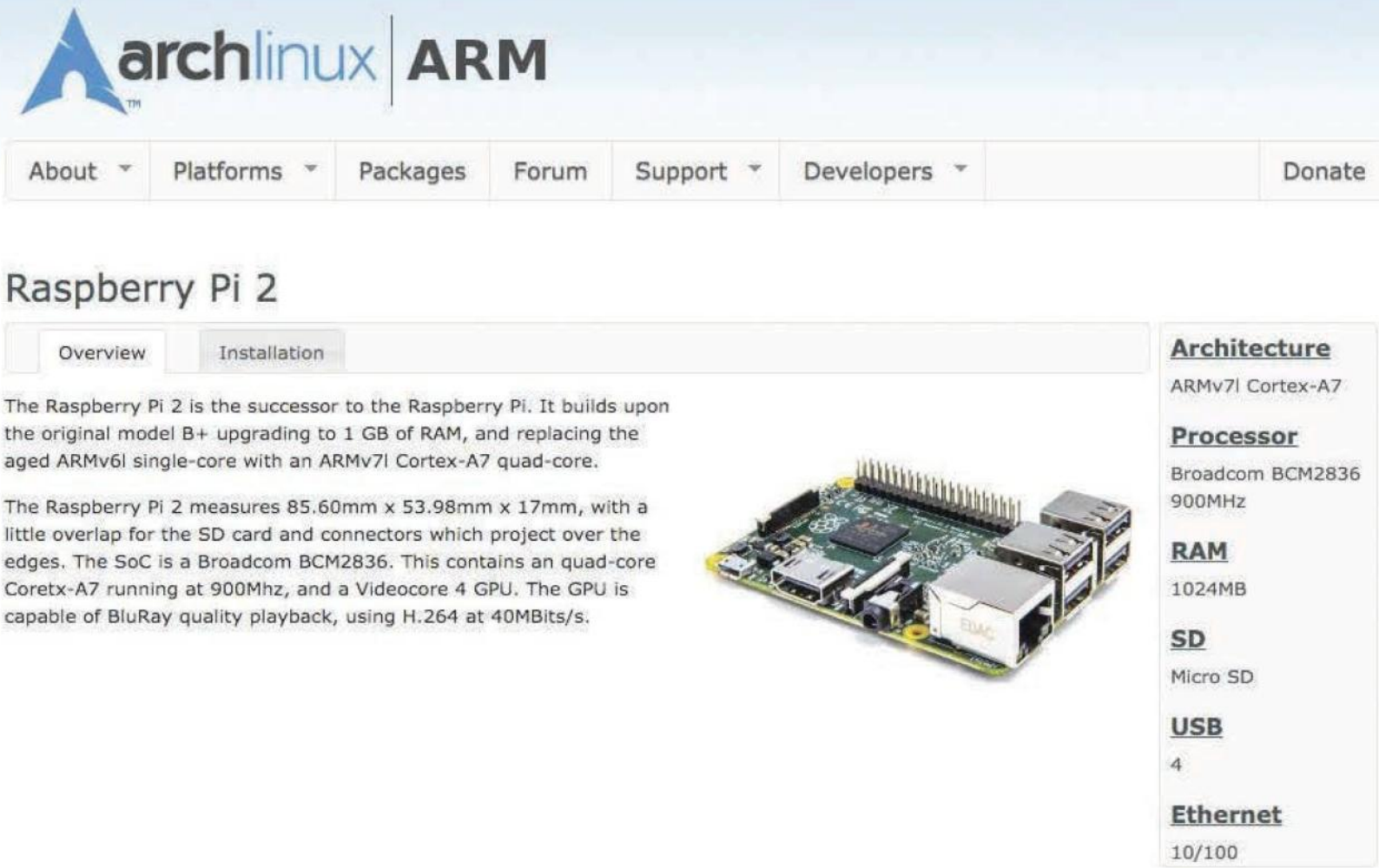


01 Install Arch Linux to an SD card

Go to Arch Linux ARM's page for the Pi 2 and click the Installation tab (http://bit.ly/1SyrGqU). You will need to carry out some manual steps on a Linux computer. Follow the instructions to download the base system tar.gz archive. Partition the card and create vfat (boot) and ext4 (root) filesystems. Expand the base system onto the card, then unmount the partitions. This will take a while as the card finishes syncing.

02 Configure the users

Once the Pi has booted up you can log in with a keyboard as root/root and then change the password. You may also want to remove the standard user account called "alarm" and create your own. Here we've used "lud" as our account name:



Above Arch Linux is an excellent choice for projects that need a lightweight, bleeding-edge software base

```
# passwd root
# useradd lud -m -s /bin/bash -G wheel
# passwd lud
# userdel alarm
```

03 Set a static IP address

Now set a static IP address so you can easily connect to each Pi without any guesswork. The OS uses systemd for service configuration. Edit the network configuration file at /etc/systemd/network/eth0. network and then reboot:

```
[Match]
Name=eth0

[Network]
```

```
Address=192.168.0.200/24
Gateway=192.168.0.1
DNS=8.8.8.8
IPForward=ipv4
```

If you would prefer to move over to a laptop or PC, you can now connect via SSH to 192.168.0.200. In our swarm there are five nodes, so the addresses range 192.168.0.200-205.

04 Install tools and utilities

Arch Linux runs on a rolling-release model, so system upgrades are incremental and packages are bleeding-edge. We will use the pacman package manager to install some essentials and upgrade the system at the same time:

```
# pacman -Syu --noconfirm base-devel wget
git sudo screen bridge-utils device-mapper
apache
```

05 Enable sudo

Configure your new user for sudo access by editing the /etc/sudoers list and then removing the comment from the line below:

```
## Same thing without a password
# %wheel ALL=(ALL) NOPASSWD: ALL
```

This enables all users in the "wheel" group to use sudo. We configured our user's primary group as "wheel" in the earlier useradd command.

"Arch Linux runs on a rolling-release model, which means system upgrades are incremental and packages are bleeding edge"

06 Clone the article's Git repository

We've put together a git repository containing some essential scripts, configuration and a pre-built version of the Docker Swarm for ARM. Log in as your regular user account and clone the repository from Github into your home directory:

```
# cd ~
# git clone http://github.com/alexellis/
docker-arm/
```

07 Install Docker 1.7.1

Docker 1.9.1 exists in the Arch Linux package system but is currently broken, so we will install the last working version and then compile it ourselves using the official build scripts:

```
# sudo pacman -U ~/docker-arm/pkg/docker-
1:1.7.1-2-armv7h.pkg.tar.xz --noconfirm
# sudo cp ~/docker-arm/pkg/docker.service /
usr/lib/systemd/system/docker.service
# sudo systemctl enable docker
# sudo systemctl start docker
# sudo usermod lud -aG docker
# sudo reboot
```

Now log in again and check that the installation was successful:

```
# docker info
```

Now we will add an exclusion to /etc/pacman.conf to stop our changes being overwritten by

Arch Linux ARM

One of the attractions of Arch Linux is that it ships as a minimal base system leaving you to decide exactly which packages you need. The boot time is much quicker than Raspbian, which has to appeal to a wider audience. The system runs a rolling release through the pacman tool, keeping all your packages up to date with the development community. Be aware that the release model updates mean that you can only install the latest version of a package.

system updates:

```
sudo ~/docker-arm/pkg/ignore_docker_
package.sh
```

08 Build Docker on Docker!

Now we have the working version, we need to compile it:

```
# cd ~/docker-arm/images/docker-arm
# ./build.sh
```

For the next 30-60 minutes a Docker development image will be set up for us, the code will be built and patched for ARM, and will then be copied into a local folder on our Pi. When the script finishes running you should get a message as below:

Created binary: bundles/1.9.1/binary/docker-1.9.1

If the output is successful then go ahead and install the changes:

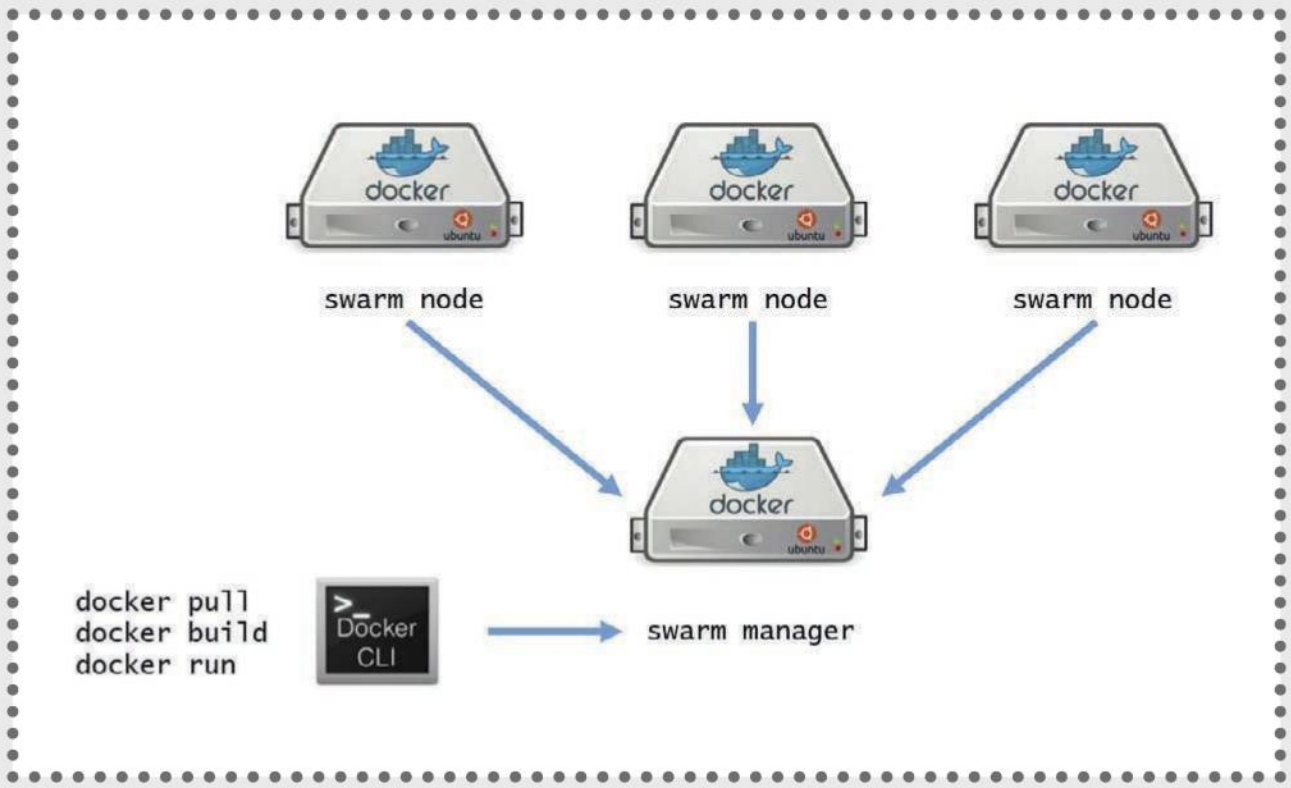
```
# cd ~/docker-arm/images/docker-arm
# sudo ./install.sh
# sudo systemctl start docker
```

09 Build Docker Swarm image

There is an official Swarm image available in the public registry, but we cannot use this because it was built for x86_64 architecture – i.e. a regular PC. So let's build our own image:

Raw materials

Here we have focused on distributing a web application across a cluster, but the Pi is also perfect for including hardware and sensing at scale. The Pi has 4 USB ports, 42 GPiO pins, audio output and a camera interface. You have all the raw materials to do something really unique. Could you extend the expressredis4.x image to light up an LED when it is processing a request, perhaps?

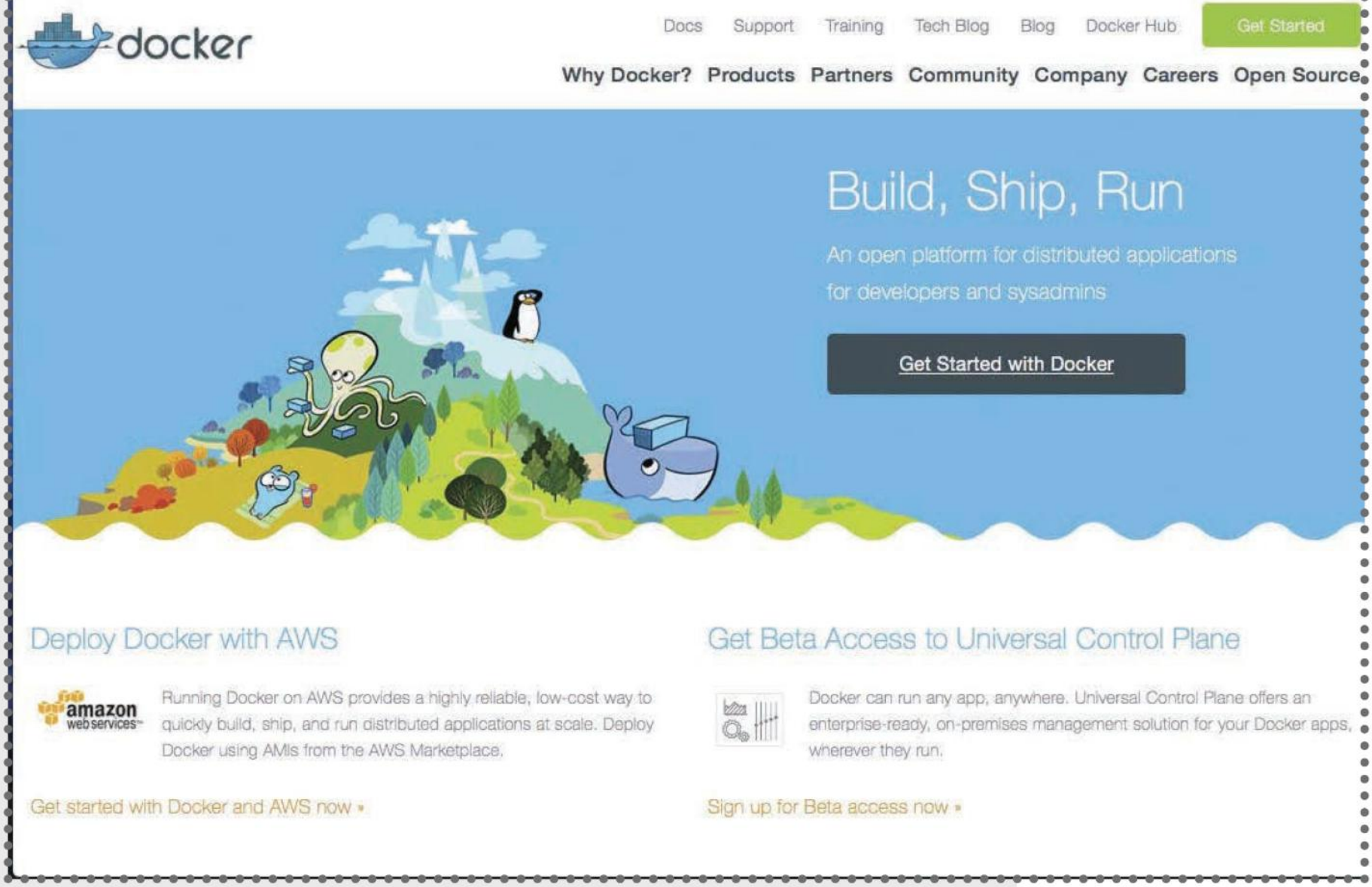



```
# cd ~/docker-arm/images/swarm-arm
# ./build.sh
# docker run alexellis2/swarm-arm
--version
```

/key.json to avoid clashes in the swarm. There are a number of additional images in the repository – you can build them as you need them as you need them or use the build_all.sh script (recommended). This could take a while to run.

10 Additional nodes

At this point you can either duplicate the SD card or run through the instructions again on each Pi. With either method, /etc/hostname and the IP address need to be updated on all Pis. If duplicating cards then make sure you delete /etc/docker/key.json to avoid clashes in the swarm. There are a number of additional images in the repository – you can build them as you need them as you need them or use the build_all.sh script (recommended). This could take a while to run.



11 Start the primary node

We are going to dedicate the first node over to managing the swarm on port 4000 and handling service discovery through Consul on port 8500. Both services will be running on the local Docker instance.

```
# cd ~/docker-arm/images/consul-arm
# ./build.sh
```

```
# ~/docker-arm/script/start_consul.sh
# ~/docker-arm/script/manage_swarm.sh
```

If you built the consul-arm container earlier, you will see that it is much quicker this time around because Docker caches the steps, so only what changes

```
alex — alex@docker1:~/repos/docker-arm/script — ssh 192.168.0.200 — 113×26
~ — alex@nuc:~/swarm — ssh nuc | ~ — alex@docker1:~/repos/docker-arm/script — ssh 192.168.0.200
[alex@docker1 script]$ docker run alexellis2/swarm-arm list consul://192.168.0.200:8500/swarm
time="2016-01-25T21:19:22Z" level=info msg="Initializing discovery without TLS"
192.168.0.201:2375
192.168.0.202:2375
192.168.0.203:2375
192.168.0.204:2375
192.168.0.210:2375
192.168.0.211:2375
[alex@docker1 script]$ export DOCKER_HOST=tcp://192.168.0.200:4000
[alex@docker1 script]$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED              STATUS
   PORTS               NAMES
910e178023dd        alexellis2/swarm-arm:latest    "/swarm join --advert"   About a minute ago   Up About a minute
   2375/tcp            docker11/join
8e6b6492c2db        alexellis2/swarm-arm:latest    "/swarm join --advert"   About a minute ago   Up About a minute
   2375/tcp            docker10/join
0bfab8377f4c        alexellis2/swarm-arm:latest    "/swarm join --advert"   About a minute ago   Up About a minute
   2375/tcp            docker5/join
01b89c22ab38        alexellis2/swarm-arm:latest    "/swarm join --advert"   About a minute ago   Up About a minute
   2375/tcp            docker4/join
6047fe5c90aa        alexellis2/swarm-arm:latest    "/swarm join --advert"   2 minutes ago        Up About a minute
   2375/tcp            docker3/join
7ba34784b8d1        alexellis2/swarm-arm:latest    "/swarm join --advert"   2 minutes ago        Up 2 minutes
   2375/tcp            docker2/join
[alex@docker1 script]$
```

between builds needs to be re-built.

12 Join the swarm

Connect to one of the nodes, i.e. 192.168.0.201, and start the auto_join_swarm.sh script. This will query the IP address of eth0 and then advertise that to consul and the swarm manager.

```
# ~/docker-arm/script/auto_join_swarm.sh
```

You will now see the swarm agent running under docker ps. Type in docker logs join if you want to see its output. Repeat this step on each of the remaining nodes.

13 Query the swarm

Log into the primary node and run the swarm-arm image passing in the address of the consul service:

```
# docker run alexellis2/swarm-arm list
consul://192.168.0.200:8500/swarm
192.168.0.201:2375
192.168.0.202:2375
192.168.0.203:2375
192.168.0.204:2375
192.168.0.210:2375
```

To start using the docker command with the swarm itself set the DOCKER_HOST environmental variable to the address of the swarm manager:

```
# export DOCKER_
HOST=tcp://192.168.0.200:4000
```

Now find out how many pooled resources we have:

```
# docker info
...
Nodes: 4
...
CPUs: 20
Total Memory: 3.785 GiB
```

14 Example: distributed web application

Let's now set up a distributed web application that increments a hit-counter in a Redis database every time we hit it. We will run several instances of this and use an Nginx load balance in front of them. We can also use Apache Bench to get some metrics. These containers need to be started in the correct order, starting with Redis, then Node and finally Nginx.

Docker Compose

Docker Compose is a tool that reads a YML file and links together containers transparently, enabling you to bring up a web service spanning more than one container and saving many keystrokes.

```
nodejs_1:
 image: node-
counter
 ports:
  - "3000"
 links:
  - redis_1
redis_1:
 image: redis
 ports:
  - "6379"
nginx_1:
 image: nginx
 links:
  - nodejs_1
 ports:
  - "80:80"
```

~ — alex@nuc:~/swarm — ssh nuc ~ — alex@docker1:~ — ssh 192.168.0.200

```
Document Path:          /
Document Length:        31 bytes

Concurrency Level:      100
Time taken for tests:   13.352 seconds
Complete requests:      10000
Failed requests:        9819
   (Connect: 0, Receive: 0, Length: 9819, Exceptions: 0)
Total transferred:      2549819 bytes
HTML transferred:       319819 bytes
Requests per second:    748.94 [#/sec] (mean)
Time per request:       133.523 [ms] (mean)
Time per request:       1.335 [ms] (mean, across all concurrent requests)
Transfer rate:          186.49 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0       7
Processing:    11  132 161.7     73     820
Waiting:       11  132 161.7     73     820
Total:         11  132 161.7     74     820

Percentage of the requests served within a certain time (ms)
  50%     74
  66%    131
  75%    166
  80%    191
  90%    358
  95%    542
  98%    665
  99%    706
 100%    820 (longest request)
[alex@nuc swarm]$ ab -n 10000 -c 100 http://192.168.0.200/
```

15 Start the redis and node containers

First start all the Redis containers, giving them names from redis_1 to redis_5:

```
# docker run -p 6379:6379 -d --name
redis_1 alexellis2/redis-arm
```

Now start an equal number of node.js containers linking them to the redis containers.

```
# docker run -p 3000:3000 -d \
    --label='node_redis' \
    --link redis_1:redis \
     expressredis4.x
```

Finally run the load balancer on the primary node:

```
# DOCKER_HOST="" docker run -d
--name=balancer -p 80:80 nginx_dynamic
```

16 Run Apache Bench

We'll start Apache Bench with 10 concurrent threads and 1000 requests in total. We started our application on six swarm agents after setting up two additional Pis.

```
# ab -n 1000 -c 10 http://192.168.0.200/
...
Concurrency Level:   10
Time taken for tests:   2.593 seconds
Requests per second: 385.65 [#/sec] (mean)
...
```

Repeating the experiment with a single Pi gave only 88.06 requests per second and took 11.356 seconds in total. You could also try increasing the concurrency (-c) value to 100.

17 Direct the swarm from your PC

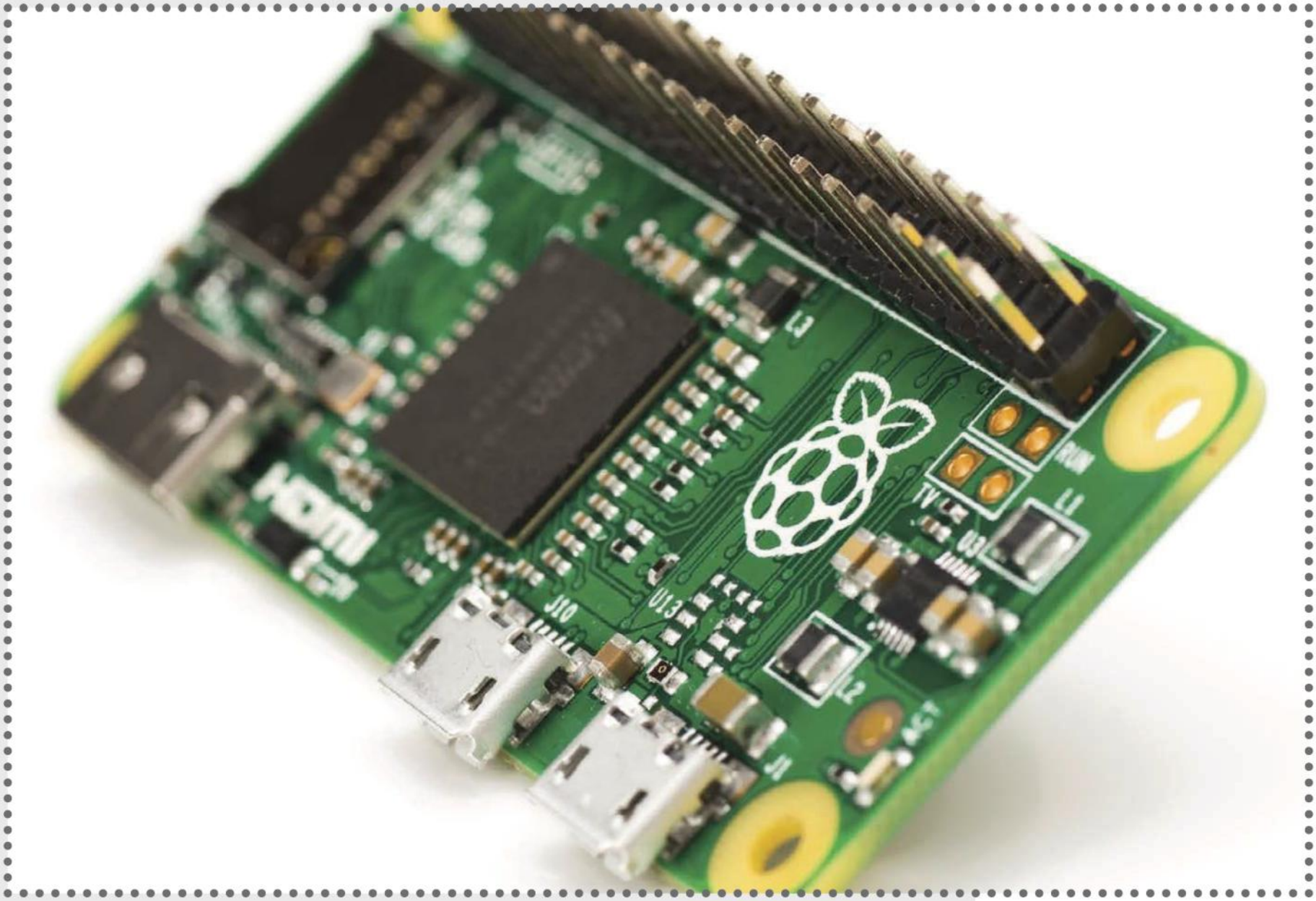
If you pull down the binary of the Docker client on its own, you can then use the DOCKER_HOST variable to point at your swarm manager, saving you from having to log into the Pis with SSH. Docker client binary releases can be found at **https://docs.docker.com/engine/installation/binaries**.

```
# wget https://get.docker.com/builds/Darwin/
x86_64/docker-1.9.1
# chmod +x docker-1.9.1
# export DOCKER_
```

```
HOST=tcp://192.168.0.200:4000
# ./docker-1.9.1 info
```

18 Wrapping up

You can repeat the steps in the tutorial until you have enough swarm agents in your cluster. One of the benefits of clustering is the distribution of work across nodes, but Swarm also provides us with linking to handle coordination between nodes. To take this idea further in your own Raspberry Pi creations, why not connect some sensors through the GPiO pins and take advantage of the Pi's hardware capabilities?



Monitoring audio

Since a Raspberry Pi is so compact, it can be used to power monitoring hardware. Learn how to use it for audio tasks

With such a small physical footprint and a low power requirement, the Raspberry Pi is a perfect platform that you can use to build your own scientific equipment. We will look at how you might be able to use your Pi to monitor and analyse sounds in your environment. This is useful if you are listening for particular sounds. You need a Raspberry Pi, some kind of USB microphone, and some kind of USB wireless connection if you want to check on the monitoring process remotely. The specifics of the hardware are up to you, but you should be able to use almost anything that is available to you. This article will focus on the Python code that you will need in order to record this audio and do some processing on it. Let's assume that you are using a Debian-based distribution on your Raspberry Pi, such as Raspbian, for the installation instructions below.

The first step is to make your microphone available to Python. PortAudio is a cross-platform library that can handle playing and recording audio on many different machines. PyAudio is the Python module that provides a wrapper around the PortAudio library. Luckily, both are available in the Raspbian package library. You can install

them with the command

```
sudo apt-get install python-pyaudio
```

The module you need to import is called 'pyaudio'. The module consists of two key objects, PyAudio and Stream. PyAudio is the object that initialises the PortAudio library and allows you to start interacting with the audio devices on your Raspberry Pi. The boilerplate code to start your program would look like this:

```
import pyaudio
p = pyaudio.PyAudio()
```

Now that you have an instantiated PyAudio object, you can open your audio device and start recording from it. There are several parameters to the open function that control recording options like the sampling rate, the number of channels, the audio format, and the size of a temporary buffer. Since you will likely need these values in the processing step of your program, you will want to store them in meaningful variables. For example:

```
CHUNK = 1024
FORMAT = pyaudio.paInt16
CHANNELS = 2
RATE = 44100
```

The open function would look like this:

```
stream = p.open(format=FORMAT,
    channels=CHANNELS,
```

```
rate=RATE, input=True,
frames_per_buffer=CHUNK)
```

By default, this function call will try and open the default audio device. If you only have one plugged in, it should do the as you expect and open that particular device. But, what do you do if there are more than one microphone plugged in? You need to add an extra parameter, named 'input_device_index', that selects the device of interest. Unfortunately, the device index that PyAudio uses is kind of arbitrary. You can use the following code to get a list of the devices and their related indices:

```
for i in range(p.get_device_count()):
  dev = p.get_device_info_by_index(i)
  print((i,dev['name'],dev['max
  InputChannels']))
```

From this output, you can find out what index value you should use in the open function call.

With an open stream connected to a microphone, you can start to read data. Since you defined the temporary buffer to be of size 'CHUNK', you use the same value in the read function from the stream. A convenient way to store this incoming data is to append them to a list. An example loop that you might want to use would look like:

```
frames = []
for i in range(0, END_TIME):
  data = stream.read(CHUNK)
```

```
frames.append(data)
```

The recorded audio now exists in the frames list, ready to be processed. If you need to keep copies of this data, you can dump it into wave files. The wave Python module lets you work with files in WAV format. You can write out the data with :

```
import wave
wf = wave.open(WAVE_OUTPUT_
    FILENAME, 'wb')
wf.setnchannels(CHANNELS)
wf.setsampwidth(p.get_sample_
    size(FORMAT))
wf.setframerate(RATE)
wf.writeframes(b''.join(frames))
wf.close()
```

"You can identify sources based on their frequencies"

… where the filename you want to use is stored in the variable WAVE_OUTPUT_FILENAME. When the audio recording portion of your program is done, you need to clean up after yourself. You need to stop and shut down the stream first, then close down the audio device you were using. This can be done with:

```
stream.stop_stream()
stream.close()
p.close()
```

Once you have real data coming in, how do you process it? The easiest thing to do is to simply plot it as a time series of the amplitude of the sound in the environment.

The matplotlib module is the go-to Python package to handle plotting and graphing of all kinds. It is a rather large module, and you will need to import whichever sub-module you need for the type of plotting you want to do. For most standard plots, you should be fine importing the following:

```
import matplotlib.pyplot as plt
```

You can then use 'plt' as the prefix for all of the plotting functions. If you just want to create a scatter plot where the 'x' values are the index of the data list, you can use:

```
plt.plot(DATA)
plt.show()
```

Here, the variable DATA has all of the sound information that you are interested in. This may be the data collected from one sampling, or a combination of many scans.
While the amplitude is one interesting quality of sound, a lot of information is lost when only looking at these values. Something that is usually of much more interest is the frequencies that exist in the sound sample. While amplitude can change quite dramatically from one scan to another, the frequencies being generated by whoever or whatever is making the sound changes very little.
So you can, theoretically, identify sources based on the spectrum of frequencies you measure. Mathematically, you can use the Fast Fourier Transform (FFT) to extract the frequencies that go into generating the sound you recorded. In Python, there are basic FFT functions in the numpy module, and more complicated sine and cosine

transforms in the scipy module. Both of these modules are huge; you can import the sections you need with:

```
import numpy.fft as npfft
```

... or...

```
import scipy.fftpack as spfft
```

You can plot these frequency spectra with:

```
import numpy as np
import matplotlib plt
import scipy.fftpack as spfft
yf = spfft.fft(DATA)
plt.plot(yf)
plt.show()
```

Here, the variable DATA contains whatever number of samples you are processing this run. This will display what the major frequencies are in your recorded sound. Once you have this data generated, you can start to make comparisons with known sound sources and potentially make identifications within the specific environment that you are monitoring.

Since all of the calls we have used above are blocking, we run the risk of getting stuck in one portion or another. To try and alleviate possible issues, you may want to look at using threads. While we are still stuck working with the GIL, it may still allow you to do more intensive processing while you are waiting for the next batch of sound data to come in. You can create a new thread with this code:

```
import thread
thread.start_new_thread(my_func,      (arg1,
arg2, arg3, )
```

Here the function 'my_func' either handles the sound recording, or the data processing functionality. Since you are going to want to pass data in and out of these functions, an easy way to handle this is with a queue object. Setting one up is as simple as:

```
import Queue
queue1 = Queue.Queue(SIZE)
```

This will then create a queue that can hold SIZE elements in it. If you hand this in as one of the parameters to your thread functions, you can use it to pass data back and forth. You just need to also create and use locks to control access to the queue to be sure only one thread is accessing it at a time. Locks can be created with this function call:

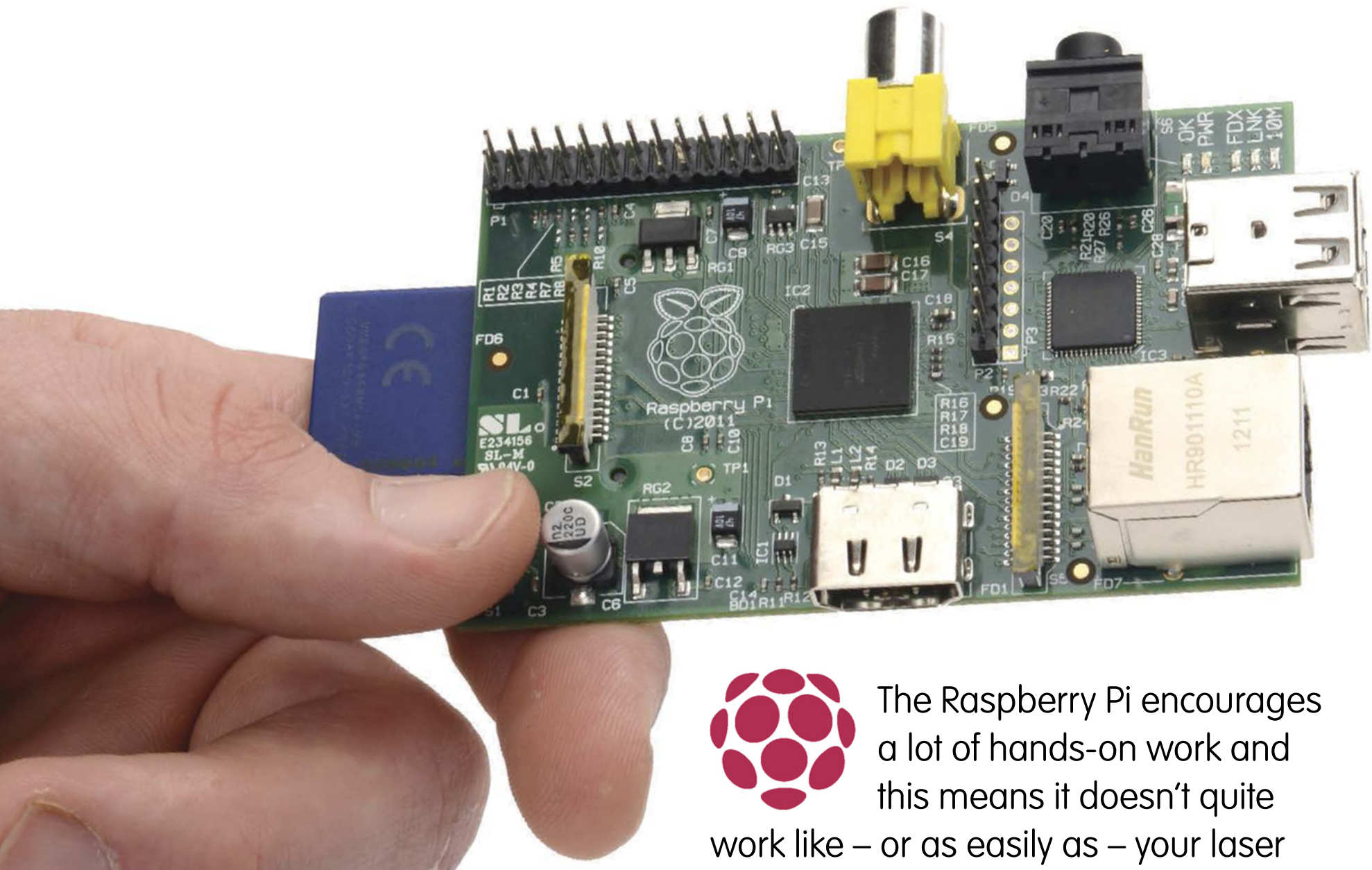
```
lock1 = thread.allocate_lock()
```

You can then use the lock methods 'acquire()' and 'release()' to manage access to the queue. You should be ready to build your own monitoring hardware for all sorts of applications.

Talking Pi

Join the conversation at...

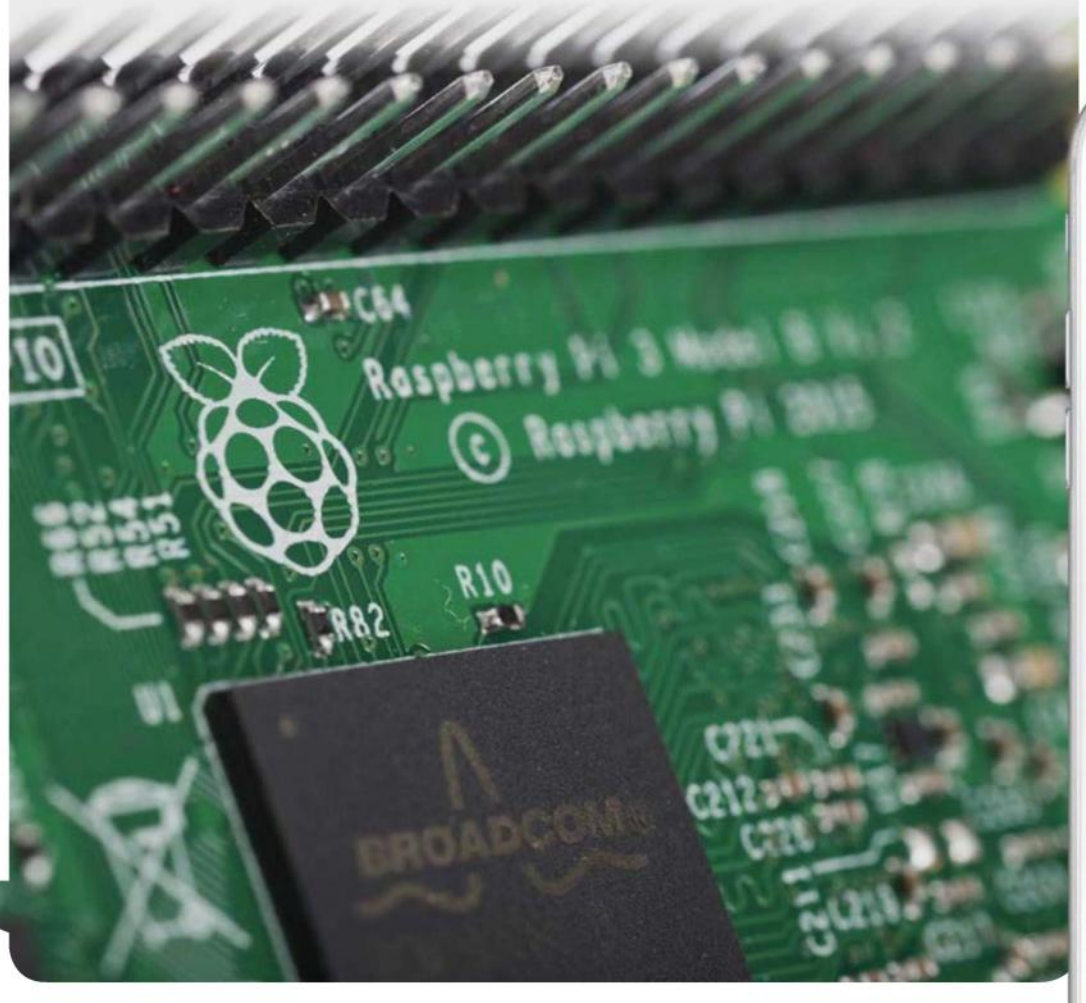
@linuxusermag | Linux User & Developer | RasPi@imagine-publishing.co.uk



The Raspberry Pi encourages a lot of hands-on work and this means it doesn't quite work like – or as easily as – your laser focus-tested smartphone interface, or even a normal computer. So we're answering your burning Raspberry Pi questions each issue – get in touch with us on Twitter, Facebook or by email.

I've just got into the Pi but something confuses me. Why do I keep hearing it called an SoC or SBC?
Cillian via email

The world of tech is full of acronyms and it can be difficult keeping up with them all, especially if you're new to the Pi and its ilk. The two terms are very similar in meaning, and both describe what the Pi is and how it works. 'SoC' means 'System on a Chip' and refers to the fact that the Pi's processor does a lot more of the grunt work than the CPU in a computer (relatively speaking; unless your PC is really old then it's got a lot more power than the Pi). The chip is responsible for a lot more of what the Pi does. 'SBC' means 'Single Board Computer', and refers to the fact that all the Pi's essential hardware lives on one tiny board!



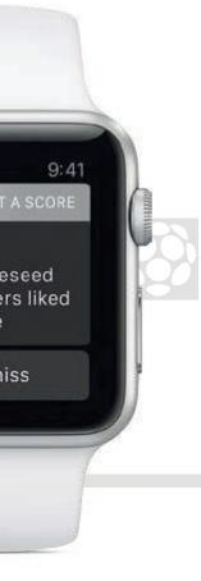
Keep up with the latest Raspberry Pi news by following @LinuxUserMag on Twitter. Search for the hashtag #RasPiMag

Should I get a Raspberry Pi or an Arduino for my projects?
Jo via email

Call us biased, but we're always going to say "Get a Raspberry Pi!" Well, we would, wouldn't we? The thing is though, it's not so much a one-or-the-other question and more a question of what you want your project to do (or what you want to do with it). Both boards have their advantages, but the best thing is that the Raspberry Pi and the Arduino can work together. Take a look at the Pi Project this issue for a great example of a maker project that combines the two!

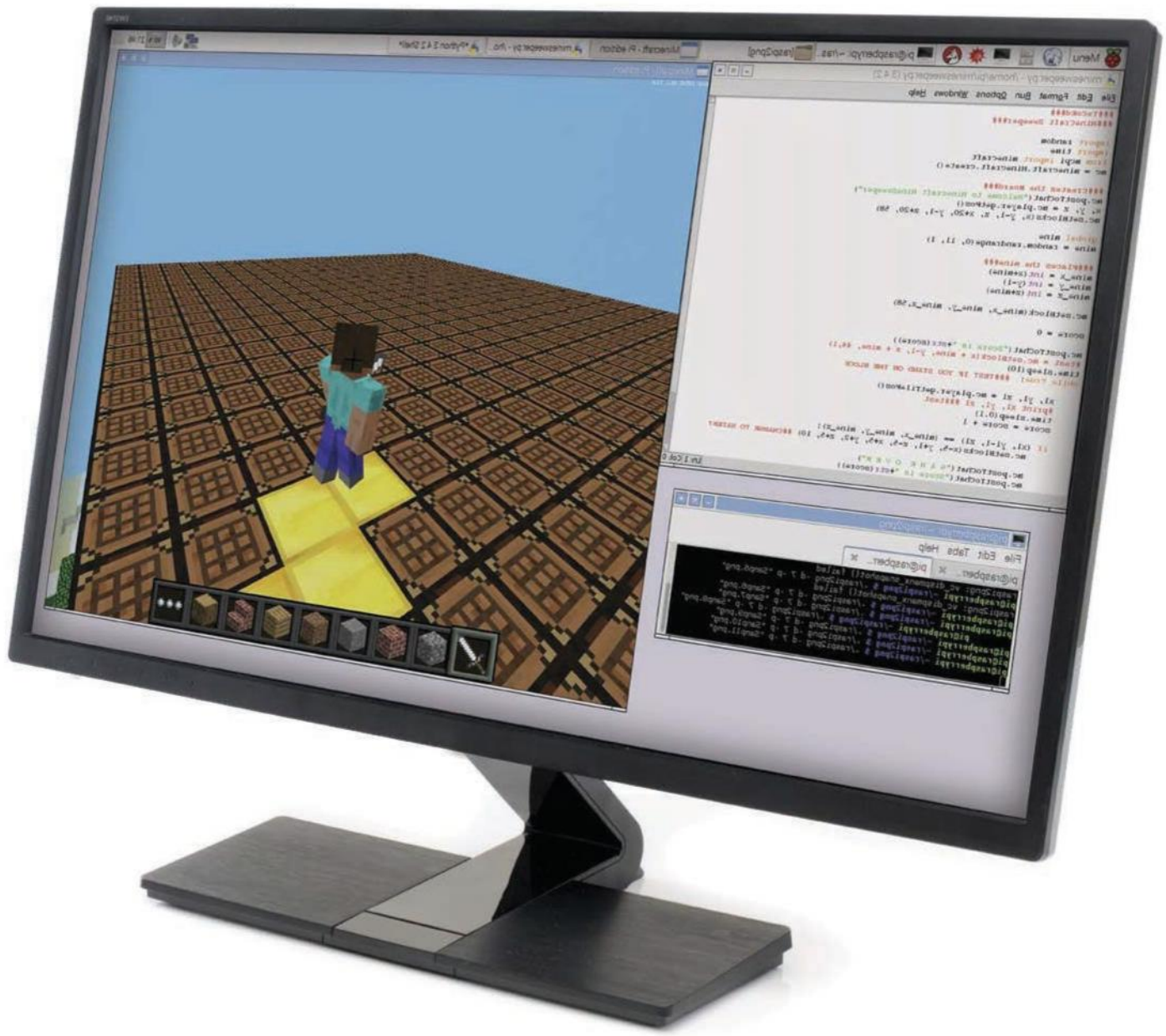
Why can't I add more RAM onto my Raspberry Pi? I have a 2B.
David via email

The whole point of an SBC is that it should have everything that you need on board already, although there are people out there like yourself who'd like more memory available for their projects. Unlike earlier models, the 2B and 3B's RAM is on separate chips on the bottom of the board, so looks temptingly like it can be upgraded. Unfortunately though, the Model 2B can't handle any more than 1GB of RAM. This is the most that the Pi's processor can handle; any more and at best you're likely to see all sorts of errors popping up, while at worst you'll seriously damage your Raspberry Pi!



Next issue

Get inspired **Expert advice** **Easy-to-follow guides**



Create a Minesweeper game in Minecraft

Get this issue's source code at:
www.linuxuser.co.uk/raspicode